UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CMU-CS-80-131	2. GOVT ACCESSION NO. A125936	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THE Cm* MULTIPROCESSOR PROJECT: A RESEARCH REVIEW		5. TYPE OF REPORT & PERIOD COVERED Interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) ANITA K JONES EDWARD F GEHRINGER editors		8. CONTRACT OR GRANT NUMBER(s) F33615-78-C-1551
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Department of Computer Science Pittsburgh, Pennsylvania 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209		12. REPORT DATE July 1980 13. NUMBER OF PAGES 222
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Air Force Office of Scientific Research (NM) Bolling AFB, DC 20332		15. SECURITY CLASS. (of this report) 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

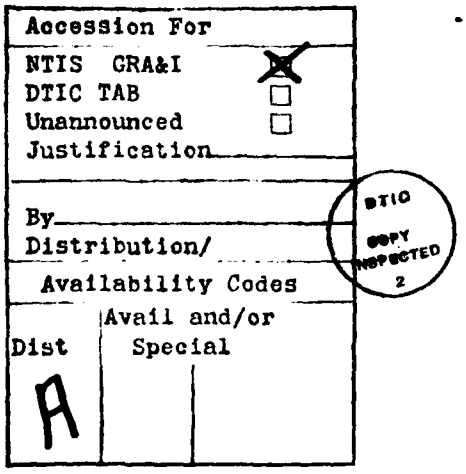
CMU-CS-80-131

The Cm* Multiprocessor Project: A Research Review

Computer Science Department, Carnegie-Mellon University

July, 1980

Anita K. Jones,
Edward F. Gehringer, editors



This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

The Digital Equipment Corporation supported the Cm* project with an equipment grant.

Table of Contents

List of Figures

List of Tables

1. Introduction

Anita Jones

The 50-processor Cm* multiprocessor became operational in fall 1979, making it one of the largest multiprocessors in existence. The associated operating systems are nearing the point that they can be used to harness the computation power of Cm* to run experimental applications. Thus, the Cm* project is nearing the end of the development phase and is entering an experimental phase.

The most important recent milestones for the Cm* project are:

- *A stable, operational, Cm*/50 configuration.* It is routinely in use by several different research groups at a time. Some enhancements to hardware continue; in particular, a prototype disk controller has just been completed.

- *Two operating systems, StarOS and Medusa, now run on multiple clusters.* They have just reached the state of beginning to support user applications and are in the process of being tuned for performance. Crucial to performance is the microcode implementation of each of the operating systems; microcode performance measurements are discussed in detail in this review.

- *Initial 50-processor application measurements.* This document reports the results of the first experiments to utilize the full fifty-processor Cm* configuration.

This review documents this turning point in our project from development to experimentation. It describes recent progress, current status, and future plans of the various research efforts that use the Cm* hardware. This review is both a "stock-taking" exercise for us, and an attempt to make available some initial results, because large multiprocessors have yet to be widely investigated—especially *quantitatively*.

Our collective, global objective is to investigate multiprocessor computation from many perspectives: architecture, operating systems design, parallel algorithm design, performance of parallel algorithms, reliability and software management. Cm* is the most recent and largest of the hardware vehicles built at Carnegie-Mellon University during a program of multiprocessor research that has now spanned more than 10 years.

This review reflects the work of a collection of people who have undertaken a variety of relatively independent research efforts. The research questions asked within the context of the different undertakings vary. The premises on which two efforts are predicated may contrast, or even be in opposition. We make some effort to contrast results, but the review, quite appropriately, does not reflect the consistency of a single research theme. And, indeed, this review provides only a "snapshot" of the state of ongoing projects.

System Development. We have developed Cm* substantially since 1977 when it existed as a 10 processor configuration, and we first wrote a comprehensive review of the project [Fuller *et al.* 77]. Since that time we have extended Cm*/10 to the Cm*/50 configuration. In addition, we enhanced or developed a number of hardware and software tools. In this document we describe these developments, particularly where they are directly relevant to the performance measurement results also given in this document.

Application performance measurement. We have experimented with a variety of applications—for example, partial differential equation solving, sorting, molecular modeling. We have had some encouraging results. And we have experimented with some parallel algorithms that are inherently not able to exploit the parallelism potentially available in the Cm* architecture to learn more about both the algorithms themselves and algorithms that are well-suited to the Cm* architecture.

For example, in the 1977 report, we presented performance measurements of alternative algorithms for a particular application: partial differential equations (PDE's). The 1977 report showed wide variance in performance for algorithms that relied on different synchronization patterns. In particular, the purely asynchronous PDE algorithm developed by Baudet was several times faster than some competitive algorithms because it does not require lock-step synchronization of iterations being performed on different processors. The purely asynchronous PDE algorithm attained *linear* speedup on Cm*/10—*i.e.*, n processors performed the PDE calculation in $1/n$-th the time that a single processor could. This was true when between one and ten processors were employed.

A key question for multiprocessor computation, in general, is what kind of speedup can be attained when substantially larger numbers of processors are employed on a single problem. Rephrased in more concrete terms, what speedup is attained, as more and more processors execute the purely asynchronous PDE computation?

Figure 1-1 shows an answer to that question graphically. It shows the performance of the purely asynchronous PDE algorithm running with three different microcodes that provide mapped addressing and basic synchronization facilities: SMAP, MEDUSA, and STAROS. At this writing, SMAP can reference memory in distant parts of the Cm* configuration faster than either of the operating system microcodes. With 35 processors, the speedup using SMAP is about 28. The speedup is not quite linear, but there is no strong downturn of the speedup curve that would indicate that any sort of limit for this algorithm has been reached on Cm*.

A more detailed discussion of this particular application is given in Chapter 5. To speak generally, the purely asynchronous PDE algorithm is reasonably well suited to Cm*/50 if the grid data is distributed across the clusters, or groups of tightly coupled processors. Better performance might

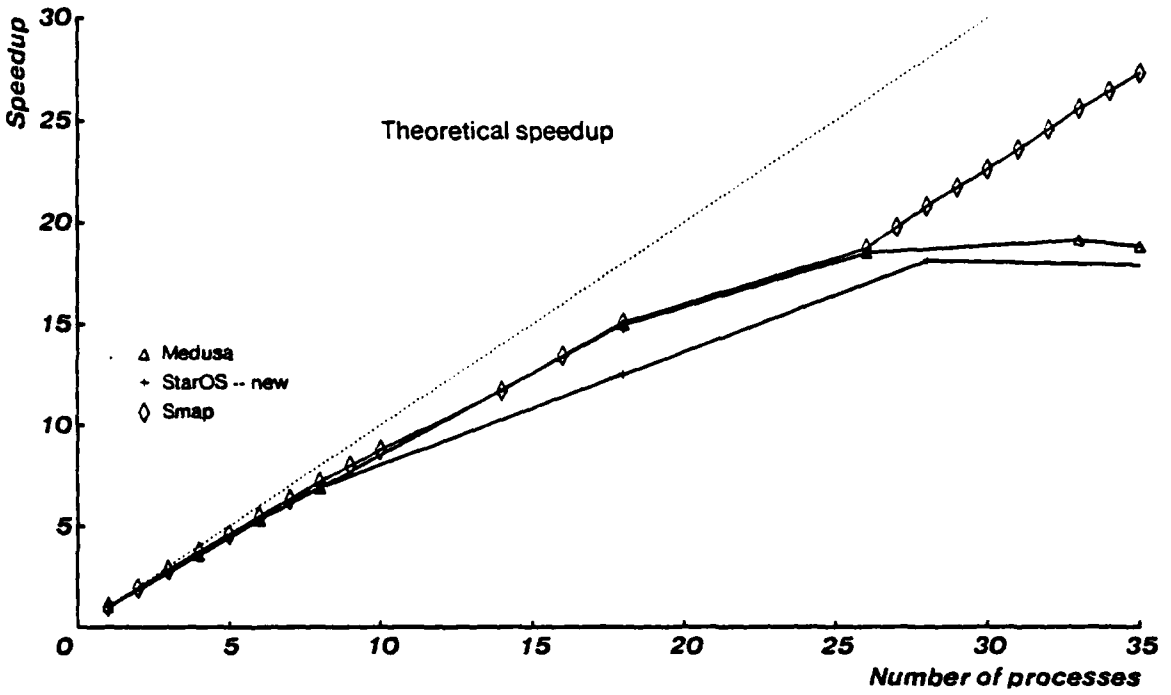


Figure 1-1: *PDE—Comparison of Three Microcodes*

result if the data were distributed across individual Cm's within a cluster, at least for very large grids. However, we have not experimented with this version of the algorithm. We have actually measured 40 per cent poorer performance in the case that the grid data is all maintained within a single cluster—in fact, within a single Cm.

Operating systems. We have developed two operating systems, StarOS and Medusa. They differ from most extant operating systems in that each—in a different way—supports some notion of strongly typed objects, and each is structured as a set of asynchronous server processes. It is the microcode implementation of certain primitives that permit the operating systems to support these attributes efficiently. For example, both systems use message communication for interprocess communication and for synchronization. In addition, both use messages to communicate requests passed between asynchronous processes in cases that uniprocessor operating systems typically use procedure activations.

To date we have written, debugged and experimented with over 10,000 words of microcode. In several chapters of this report we discuss measurements and experiments with the use of the various microcodes. In particular, Chapter 10 discusses microcode performance measurement, for example, the cost of mapped references and message communication operations.

We contrast the cost of mapped references for three microcodes, SMAP, which uses the simplest possible virtual addressing scheme that supports cross-cluster addressing; MEDUSA, which supports two-level descriptor-based addressing; and STAROS, which utilizes three-level, or capability-based, addressing. Mapped references require assistance from a processor called the Kmap. Because the speed of mapped references is a first-order determinant of how fast an application can execute, we have measured the average time for a memory reference to be completed, once a processor has placed a 16-bit address on the bus. The cost of an in-cluster mapped reference for the three microcodes is:

	Kmap cost
SMAP	8.3 μseconds
MEDUSA	8.3 μseconds
STAROS	8.6 μseconds.

The average time for a memory reference emanating from one cluster to be serviced by a distant, but directly connected, cluster is

SMAP	26.2 μseconds
MEDUSA	30.8 μseconds
STAROS	35.3 μseconds.

The above measurements assume no contention. In addition, they assume that any caching of needed data structures within the data RAM of the Kmap has been performed prior to measurement. One observation of interest is that the dynamic cost of capability-based addressing (STAROS) and descriptor-based addressing (MEDUSA) is roughly the same for the PDE application as can be seen in Figure 1-1. This contradicts the belief held by many that capability addressing has excessive dynamic cost compared to other kinds of addressing mechanisms, such as segment-descriptor or simple paged addressing.

Conclusion. This review provides an overview of the set of research efforts that collaborate in their use of the Cm* hardware. For the reader who wishes to delve more deeply into Cm*-related research, additional papers and technical reports are listed in in the Appendix I. The following references are particularly recommended:

[Fuller *et al.* 77] S. H. Fuller, A. K. Jones, I. Durham, eds.
Cm review*.
Technical Report, Computer Science Department, Carnegie-Mellon University, June, 1977.

[Fuller *et al.* 78] S. H. Fuller, J. K. Ousterhout, L. Raskin, P. Rubinfeld, P. S. Sindhu, and R. J. Swan.
Multi-microprocessors: An overview and working example.
Proceedings of the IEEE 66(2):216 – 28, February, 1978.

[Jones *et al.* 79] A. K. Jones, Robert J. Chansler, Jr., I. Durham, K. Schwans, and S. R. Vegdahl.
StarOS, a multiprocessor operating system for the support of task forces.
In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 117 – 27. ACM/SIGOPS, Pacific Grove, California, December 10 – 12, 1979.

[Ousterhout *et al.* 80]
J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu.
Medusa: an experiment in distributed operating system structure.
Communications of the ACM 23(2), February, 1980.

2. The Cm* Hardware

This section provides a brief overview of the structure and components of the basic Cm* hardware. More detailed descriptions may be found in a number of earlier publications: the original description of the design and implementation of Cm* appears in two papers presented at the 1977 National Computer Conference [Swan *et al.* 77a, Swan *et al.* 77b]; the design of the switching structure of Cm* and a detailed account of one particular addressing structure are presented by Swan [Swan 78]; a discussion of Cm* in the context of multiprocessors in general appears in [Fuller *et al.* 78]; an evaluation of the performance of the hardware is the subject of [Raskin 78] and [Fuller *et al.* 77].

2.1. The Structure of Cm* *Pradeep Sindhu*

Cm* consists of fifty processor-memory pairs called computer modules, or Cm's,[1] connected together by a hierarchical, distributed switching structure (Figure 2-1). The lowest level of the switching hierarchy consists of **Slocals**, local switches that connect individual Cm's to the rest of the structure. Cm's are grouped together into **clusters** that are presided over by high-speed micro-programmable communication controllers called **Kmaps**. A Kmap provides the mechanism for Cm's in its cluster to communicate with each other and cooperates with other Kmaps to service requests from its Cm's to access Cm's in non-local clusters. Since Kmaps are microprogrammable, it is usual to implement key operating system functions in the microcode of these processors in addition to the normal functions of address mapping. All communication mediated by the Kmaps is implemented via packet switching rather than circuit switching to avoid deadlock over dedicated switching paths. Packet-switched communication also allows the processing of requests by the Kmaps to be overlapped since switching paths are no longer allocated for the duration of a request; this leads to considerably better utilization of the switching structure. The inter-connection structure of Cm* at the level of clusters is essentially arbitrary. The Kmap of each cluster has two bi-directional ports each of which may be connected to a separate **intercluster bus** to implement a variety of inter-connection schemes. In the current configuration of the hardware, all five Kmaps are connected to both of the intercluster busses as shown by Figure 2-1.

The modularity and distribution present in the hardware provides the potential for building a system that can tolerate failures within its components. There is no central point of failure in the structure of

[1] After the PMS notation of Bell and Newell [Bell 71].

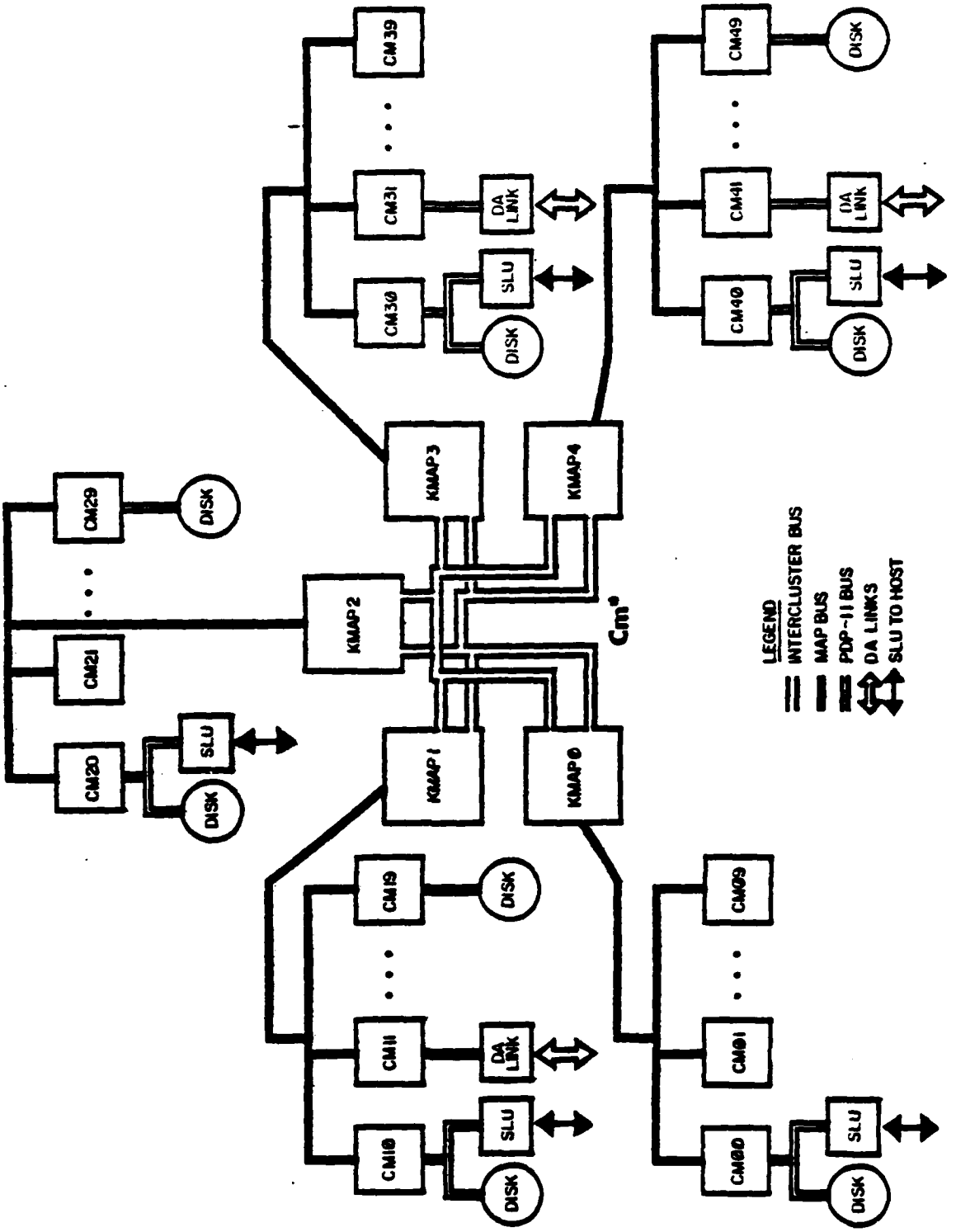


Figure 2-1: The Structure of Cm*

Cm*, and there are no *static* master-slave relations between its components.[2] The main memory of the system consists entirely of the local memory of all the Cm's; there is no central, shared memory. The processors in the system are individually slow but there is a large number of them and they are distributed physically. As a result, errors occuring in one processor ought to be localizable to that processor, and the loss of a single processor ought not to degrade the operation of the system substantially. The communication resources of the system, the Kmaps, are individually more powerful and are also physically distributed. However, since there are fewer Kmaps, and since each Kmap presides over a larger portion of the system than a processor, the loss of a Kmap will have a greater impact on the system than the loss of a processor.

There are two attributes of the architecture that combine to make Cm* unique for the purposes of experimenting with multi-computer systems. First, the computational and communication resources of the system are physically distributed so that problems that are encountered in the context of Cm* are likely to apply in the broader context of distributed systems. From any one point in the system a small set of resources is accessible at low cost. The rest of resources of the system are also accessible, but with greater overhead. Second, the presence of powerful, programmable communication controllers permits a wide variety of systems ranging from closely coupled shared memory to pure networks to be evaluated on the same hardware.[3] This combination of distribution and sharing is enhanced by the particular way in which processors are interfaced to the rest of the system. Since the Slocal of a processor cuts the addressing path between processor and memory, the method of making ordinary memory references appears to the processor to be independent of the location of the memory. Moreover, highly complex functions implemented within Kmap microcode may be invoked exactly like memory references by giving special meaning to particular portions of the processor's address space.

2.2. The Components of Cm*

The basic computing element in Cm* is a processor-switch-memory combination called the Cm. Each Cm consists of a standard off-the-shelf DEC LSI-11 processor, 64 or 128K bytes of memory, one or more I/O devices, and a custom-designed Slocal which connects the processor-memory combination to the rest of the system (Figure 2-2).

---

[2] Even though there are no static master-slave relationships between components, temporary master-slave relations do exist between components over the lifetime of a given request.

[3] To date at least six different systems spanning the entire range from shared memory to fast message systems to networks have been implemented on the Kmaps.

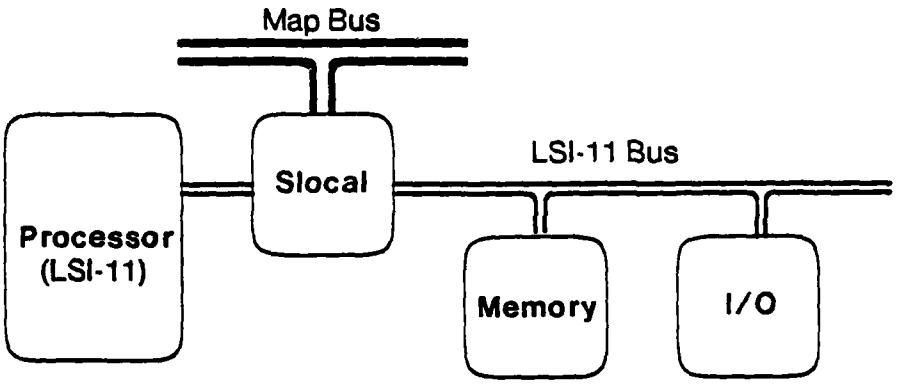


Figure 2-2: Details of a Computer Module

When the processor of a Cm initiates a memory reference, the Slocal of that Cm is responsible for determining whether the reference is to be directed to local memory or out to the Kmap for further mapping. As shown in Figure 2-3 the Slocal uses the four high-order bits of the processor's address along with the current address space to access a mapping table which determines whether the reference is to proceed locally or not. References that map to local memory proceed with no loss of performance; references that map to another Cm in the same cluster as the referencing Cm are slower by a factor of three; and references that map to a Cm in another cluster are slower again by a factor of three. These figures are the best possible ratios that can be achieved on Cm*, and therefore correspond to constraints imposed by the hardware itself rather than to implementation quirks of any particular microcode for the Kmaps. All I/O devices in Cm* are connected to the various LSI-11 busses. Since there is no inter-processor communication mechanism other than the standard one for memory references, interrupts generated by an I/O device must be fielded by the processor to which the device is directly attatched.

The other important component in the architecture of Cm* is the Kmap. The Kmap is a fast (150-ns cycle), horizontally microprogrammed (80-bit wide) microprocessor that provides the basic address mapping, communication and synchronization functions in the system. The flexibility and power of the Kmaps is in part responsible for the variety of systems that have been implemented for Cm*. Since the Kmap is microprogrammable, it is possible to move key operating system primitives into the Kmap and experiment with a number of operating system designs that differ in their lowest levels. Later chapters of this report contain detailed descriptions of four of the largest microcode systems that have been written for the Kmaps.

The Kmap itself consists of three tightly coupled processors (Figure 2-4). The bus controller, or

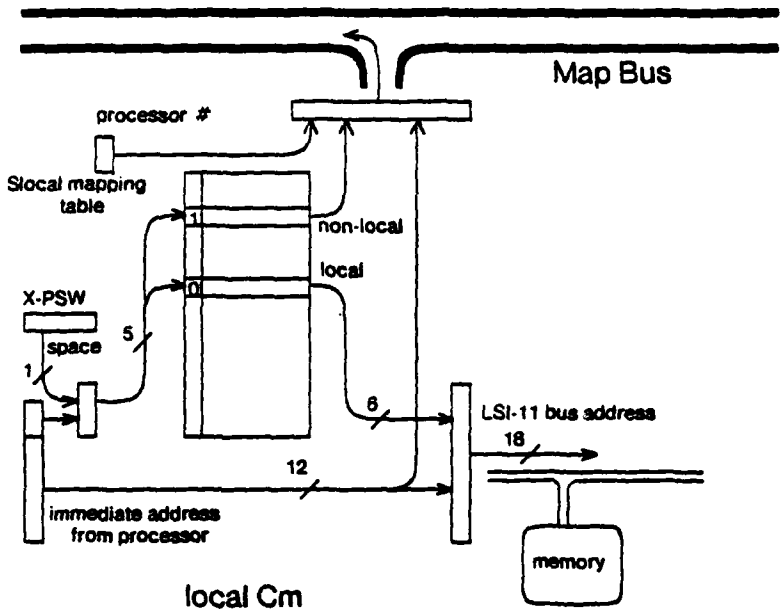


Figure 2-3: Address Mapping in the Slocal

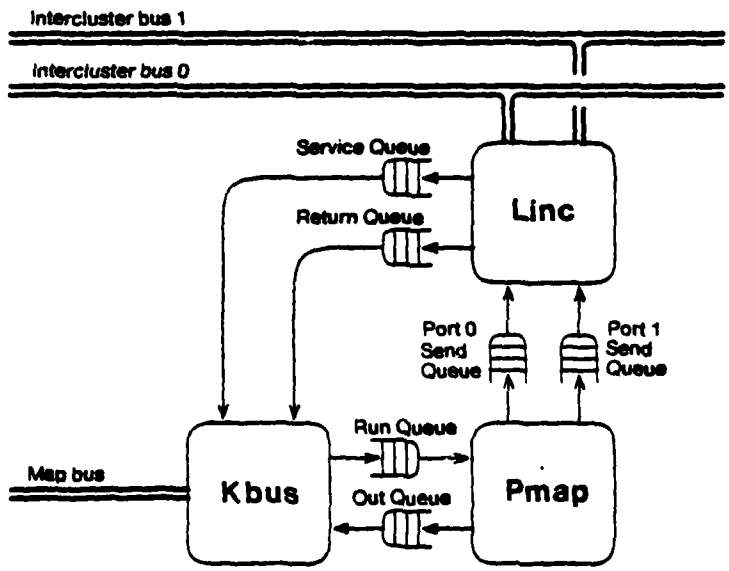


Figure 2-4: The Components of the Kmap

Kbus acts as the arbitrator for the bus that connects Cm's in the local cluster to their Kmap; the

Linc manages communication to and from the Kmap to other Kmaps; and the mapping processor, or Pmap responds to requests from the Kbus and Linc, and performs most of the actual computation for a request for service. The Pmap also directs the Kbus and Linc to perform any needed operations on behalf of the request being processed. Since the Kmap is much faster than the main memory of the LSI-11's, most of the time for a simple reference is spent in accessing memory and relatively little time is spent within the Kmap. The throughput of processing requests could therefore be increased considerably if the Kmap is allowed to process more than one request concurrently. In fact the Kmap is hardware multiprogrammed to a degree of eight, permitting it to handle up to eight requests at any one time. Each of the eight partitions of the hardware is called a **context** and key state for each context is duplicated to make context swaps efficient.

2.2.1. Present Hardware Configuration *Lawrence Butcher*

The original Cm* configuration contained 10 Cm's and 3 Kmaps. The results of earlier experimentation with it were reported in 1977 [Fuller *et al.* 77]. In the last two years the Cm* multiprocessor has grown from a 10-Cm machine to a 50-Cm machine with 5 Kmaps. This section serves to describe the present hardware configuration and explain how it has changed since the previous report.

As the additional hardware was built and brought up, a series of modifications were made to all components of the machine, and at present the entire Cm* machine is stable and free of known bugs. A large number of peripheral devices were added to the system as the machine grew. At the present time, the Cm* configuration contains the following components:

- **Cm* support hardware.** A PDP-11/10 system cross-patches user terminals to resources on Cm*. It runs a software system known as the Cm* Host, which is described in detail in Section 3.1. The Host is much the same as it was in 1977. The number of lines to the Computer Science Department's Front End computer has increased from 2 to 3, and there are now 4 local terminals, two of which are 9600-baud CRT's. There are additional lines available for hardware development and debugging.

- **Kmaps.** 2 Kmaps were built to bring the total number of Kmaps to 5. The writable control store on each Kmap was quadrupled in size to the present 4K eighty-bit instruction capacity. The number of Kmap general-purpose registers and the number of subroutine-return registers were increased from 8 to 32 each. There are two **intercluster busses**, which serve to transmit messages from one Kmap to another. Each Kmap is connnected to both intercluster busses. Normally all 5 Kmaps are interconnected to form a 5-cluster system; however, for performance evaluation it is possible to rearrange the processors to run with all Cm's in a 4-cluster system. In addition, 5 **map bus monitors** were constructed to monitor communication on the bus between Kmaps and Cm's. These monitors serve as multiprocessor front panels and are capable of producing trigger signals on selected Map Bus events. They have been used to obtain accurate hardware performance measurements and are used in hardware debugging.

- **Cm's.** 40 additional Cm's were assembled to make a total of 50 Cm's. All Cm's now use DEC 64K-byte memory cards; 34 Cm's have 64K bytes of memory and 16 Cm's have 128K bytes of memory. All Cm's have line-time clocks. There are **serial-line connections** from the Cm* Host to ten individual Cm's. These connections permit programs or data to be loaded directly from the Host to these 10 Cm's, two of which are in each of the 5 clusters. Cm's lacking serial-line connections have to be loaded from other Cm's via a Kmap, or from peripheral devices to which they are attached.

- **Peripherals.** The Cm* processor has access to a large number of peripherals in addition to the 10 serial-line connections to the Cm* Host. They are—

  1. *Two 4800-baud serial-line connections* switchable between the Computer Science Department's two DEC KA-10 computers.

  2. *One 4800-baud serial line* to a vector graphics terminal. This has been used in experiments such as hidden-line elimination (Section 6.3).

  3. *One 32-bit real-time clock.* This device has 0.5 μsec. resolution and uses a crystal timebase. The counter can be set to zero under program control to simplify interval measurements. This clock can accurately measure single instruction-execution times.

  4. *Four DEC KUV11-AA Writable Control Store boards.* These boards allow user microcode to be executed on modified LSI-11's. The LSI-11's behave like ordinary processors until the WCS board is enabled. These boards have been used by several classes to teach microcoding and could be used for standalone Cm* projects. At present there are no plans to make these devices available to users of the various Cm* operating systems, as a microprogram with bugs could cause the LSI-11 to ignore all interrupt and halt requests.

  5. *Four DA-Link boards.* These high speed DMA devices communicate over an 18-bit parallel bus with a similar device on the Computer Science Department's DEC KL-10. Between an unloaded LSI-11 and an unloaded KL-10 these links can transfer in excess of six hundred thousand words per second. Designed and built by the department, these devices have greatly decreased the time needed to complete a compile-link-download cycle. The file-transfer system based upon them (Section 3.6) provides programs executing on Cm* with access to both files and programs on the department's KL-10.

  6. *Two DEC RP-11 disk controllers.* Locally accessible disks will provide the main file storage for the Cm* operating systems. These 2 disk controllers are interim devices that will be replaced by controllers built in-house (Section 2.2.2.) Each of these controllers can control eight RP03 40-megabyte drives, although each will probably be used with four or five drives.

  7. *Two Xerox ETHERNET interfaces.* These boards provide a DMA interface to the 3-megabaud ETHERNET, a network to which most machines in the department presently have access. Eventually there will be access to the Arpanet through the ETHERNET.

  8. *One character-mapped console display terminal.* An ADM-3A terminal was modified

so that it refreshes from a dual-ported memory which connects directly to an LSI-11 bus. Each character on the screen is readable and writable as a byte of LSI-11 memory. The terminal is therefore addressable by all 50 Cm's. The various operating systems can provide protected access to portions of the screen. When several Cm's use the display simultaneously, it can be updated at speeds of several hundred-thousand characters per second.

- **Diagnostic Processor.** A Diagnostic Processor (DP) has been added to the Cm* system to collect hardware reliability and availability information. It hosts a program called the AutoDiagnostic Master which runs diagnostics on those Cm's that are otherwise idle. The DP is an LSI-11 with 28K words of memory. It has 2 serial-line connections to the Cm* Host. One connection provides a user interface through which one can request status reports about particular Cm's, particular clusters, or of the entire Cm* processor. The second serial line interface is used by the AutoDiagnostic program as a command interface to the Host. The program logs in over the second line, directs the Host to run diagnostics for it, and on operator request transfers a statistics file to the department's KL-10.

- **Hooks Processors.** In addition to its 50 Cm's, Cm* has three LSI-11's, known as **Hooks processors**, which are used for debugging the Kmaps. These processors control the **Hooks**, which is the collective name for a set of hardware which has been designed into each Kmap in order to permit complete external control and diagnosis of it. The Hooks consist of several control registers and other hardware within the Kmap, an LSI-11 bus interface to make the hardware accessible from an LSI-11, and a bidirectional **Hooks bus** used to transmit information between the control hardware and the LSI-11 bus interface. The Hooks appear to an LSI-11 as a group of eight words in its physical address space. By reading and writing these words, the Hooks processor has almost complete control over the internals of the Kmap. It may load microcode; start, stop, and single-cycle the Pmap/Linc and Kbus clocks; read out most and write some of the internal registers of the Pmap; disable certain error checks within the Pmap; and initialize the Kmap.

2.2.2. Disk Controller

Pradeep Reddy

A prototype disk controller has recently been designed and built for Cm*. The goals of the project were:

- The controller should be capable of supporting both RP03 and RP06 disk drives. The RP06 has higher transfer rates and more storage. The disk controller has to be capable of cycle times fast enough to keep up with the faster drive.

- The controller should be both inexpensive and compact. This is directly in keeping with the philosophy of Cm*. The disk controller must fit on a single Hex wire wrap card that would plug into the LSI-11 bus.

- The controller should make both hardware and firmware debugging easy.

The motivation for the project came from the ten RP03 disk drives that were part of the DEC hardware grant in the summer of 1978 and the fact that Cm*, being a distributed system, has a potentially large

I/O bandwidth. To realize this bandwidth in terms of Disk I/O it is necessary to distribute disk drives across the system say one or two per cluster. In this way all the disks could conceivably be performing I/O simultaneously. Thus it becomes necessary to have many disk controllers which are both compact and inexpensive.

None of the commercially available disk controllers fulfilled all the requirements and most were too expensive in the quantities that were needed. Hence it was decided to build a disk controller in house. The project entered its design phase in February 1979.

Since the controller must be capable of using both RP03 and RP06 disk drives, all the drive-dependent circuits were moved from the controller to an **adapter board** that would be tailored to meet the varying specifications of the two drives. The adapter board consists of data-acquisition, data-generation, clock-generation and line-driver circuits.

To achieve upward compatibility the controller had to be designed for the faster transfer rate, 6.5 MHz, of the RP06. A more critical speed constraint was the decision to provide capability for error correction on the controller. The scheme involved the addition of 10 words of error-correction codes to the end of each block of data. This enables the controller to correct burst errors up to 16 bits long. Though this solution is somewhat expensive in storage, all the code generation and checking could be done in microcode. To perform this in real time the microprocessor has to be capable of 13 cycles for every word that is transferred to or from the disk. A microcycle time of 190 ns is necessary for the RP06.

The current implementation of the disk controller is a microprogrammable machine. Its main constituents are a bit-slice ALU capable of cycling at about 190 ns, a 1K x 40 writable control store and a 1K x 16 data RAM for internal storage. It has FIFO queues to provide data buffering between the internal data bus and both the disk and the LSI-11 bus. The disk data-acquisition logic can operate up to 10 MHz. About 16 parallel lines are driven off-board to issue commands to the drives and about 12 lines can monitor the status of drives. There is considerable hardware support for hardware and firmware debugging. The disk controller is capable of using up to 8 drives but the current design of the RP03 adapter card can handle only four drives. In the Cm* system it is hoped that no more than one drive will be connected to a controller to maximize system throughput. The current implementation for an RP03 cycles at 300 ns, which is quite adequate for the slower bit rate. The same hardware, with faster control store RAMs (about 50 ns access time) will yield the 190 ns cycle needed for the RP06. Figure 2-5 diagrams the the data paths in the disk controller.

Though the controller was meant to be used as a disk controller the interpretation of all the "disk" registers is soft. The device-dependent circuitry is also happily off board. We can view the controller as being able to drive some parallel lines (about 16) and monitor the status of some more lines (about

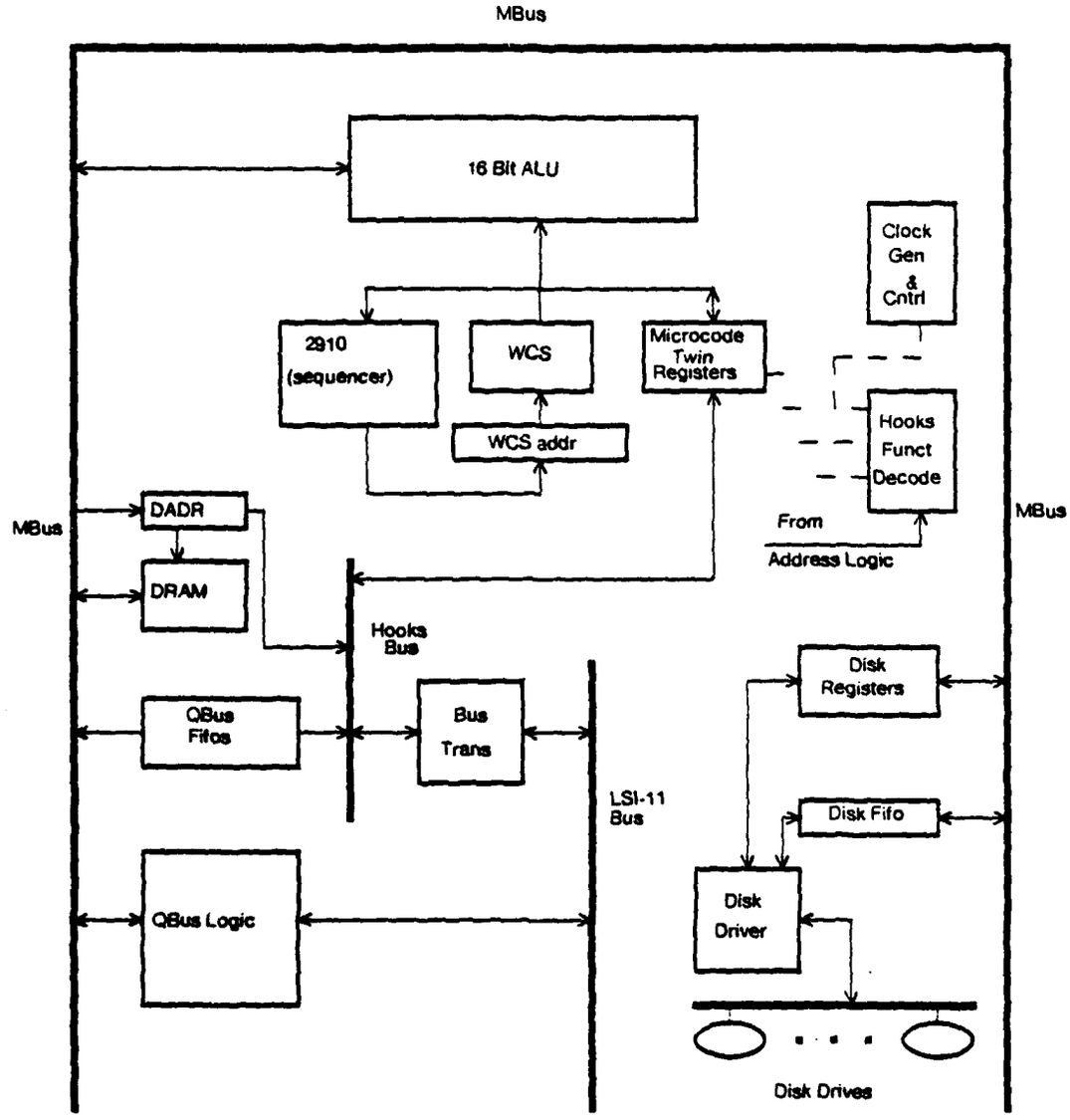


Figure 2-5: Data Paths in the Disk Controller

12). The frequencies at which these lines are driven or monitored is as fast as the cycle rate of the micromachine. The device also drives a serial data line with frequencies up to 10 MHz, and can acquire data from another at the same maximum frequency.

The adapter card that goes between the controller and the device will then tailor these signals to the needs of the device. The controller can be made to work with a new device by building a new adapter card, which is quite trivial, and writing new microcode, which is not so trivial. Most serial devices with data rates of up to 10 MHz can be handled by the controller. The controller could quite easily be turned into a network controller.

A prototype disk controller has been designed and fabricated. A debugger was written before attempting to bring up the hardware. The debugger proved to be of tremendous help during this phase. An RP03 adapter card was then fabricated to interface the controller to the Cm* drives. Microcode, making the controller appear like the DEC RP11 disk controller, was developed and data transfers between the drive and the controller are now possible.

The next step is to determine a programmer interface to the controller and implement this interface in microcode. Cm* could use many copies of this disk controller to improve both its I/O capability and bandwidth. Some of the controllers could be driving disks and others could be interfaces to local networks. Hardware work to enable the controller to interface to a network, however, needs to be done. It will also be worthwhile to investigate the possibility of turning the controller into a PDP-11 device. This seems eminently possible and desirable in the CMU environment.

Pradeep Reddy was the principal designer of the disk controller. Richard Swan, John Ousterhout, Pradeep Sindhu, and Lawrence Butcher originated many design ideas and helped develop them. Butcher designed and fabricated the RP03 adapter card.

2.3. Reliability Studies on Cm*

Michael Tsao

The 50-module Cm* system provides a unique opportunity for gathering data about computer failures, both hard and transient. Cm* hard-failure data was used in a reliability comparison between industrially produced components and CMU-built one-of-a-kind components. It was found that CMU-built components, which did not use burned-in parts, generally have a higher failure rate. It was also shown that, as expected, the distribution of hard failures follows the exponential distribution. Cm* transient error data was also analyzed for the distribution of interarrival times. It was found that the distribution follows a decreasing failure-rate Weibull function. This is at variance with the standard assumption that transient errors exhibit an exponential distribution with a constant failure rate.

2.3.1. Hard Failures

Hard failures, or permanent faults, are continuous and stable, reflecting an irreversible physical change in the hardware. Cm* hard-failure data, collected from the engineering log book, was used to calibrate existing reliability models. The Mean Time Between Failure (MTBF) was calculated assuming failures were independent. The MTBF was obtained by dividing the total time by the total errors. In order to combine data for all of the computer modules (Cm's), a concept called "module time" was introduced. If there are several modules running during a period of time, then the module time is the sum of the time that each module was up. Module time is divided by the total number of recorded failures for the entire multi-module Cm*. This yields the MTBF for a single "typical" module. Because of the small number of failures per computer module, it is a more realistic reliability measure than the MTBF for any particular module. Table 2-1 presents this module-time data and the MTBF, measured in module hours, for Cm*.

Table 2-1: Cm* Hard-Failure Data From February 1977 to May 1978

Component	Complexity (Chips)	# of Modules	Total Time (Mod. Hrs.)	Total Failures	MTBF (Mod. Hrs.)
Kbus	138	3	36696	8	4587
Pmap	106	3	37416	12	3118
Control store	116	6	68328	4	17082
Data RAM	142	3	37082	2	18540
Linc	116	3	22608	0	—
DEC LSI-11	68	14	163200	10	16320
Slocal	126	10	120720	5	24144
4K memory	56	21	260568	5	52003
16K memory	104	10	122280	5	24456
SLU	28	17	223248	5	44650
Power board	6	16	195456	3	65152
Refresh	14	16	162912	0	—

The "complexity in chips" mentioned in the table is a measure of the actual number of chips used in each component. In the case of the Digital Equipment Corporation (DEC) LSI-11, the actual number of chip sockets used is 76; of these, 72 contain digital integrated circuits. Since unused functions on the chips add up to the equivalent of 4 unused chips, the number of chips used is recorded as 68.

An ANOVA (analysis of variance) on the error-log data shows that uncertainties associated with module commissioning dates (*i.e.*, initial power up and integration into the system) were insignificant.

The failure distribution was shown to follow the exponential distribution, *i.e.*, a constant failure-rate Poisson process used in the Military Standard Handbook (MIL 217B) *reliability model* [Siewiorek *et al.* 78b].

MIL model 217B assumes that the failure of electronic components is a Poisson process and the failure distribution follows the exponential distribution, which is characterized by a constant failure rate over time. The failure rate for a single IC chip can be predicted using the following MIL 217B model parameters: a learning factor based on the maturity of the production process, a quality factor based on the procedure for incoming screening of components, the ambient operating temperature, a factor based on the benignity of the operating environment (considering such factors as humidity and vibrations), and two complexity factors based on the number of random logic gates, and the number of memory bits in the component.

Cm* chip-failure data and vendor data was used to calibrate the MIL 217B model, and was then compared with its predictions. The model turned out to be too pessimistic by a factor of 16 to 64, compared to the Cm* data. That is, it predicted 16 to 64 times as many failures as actually occurred. One can speculate that MOS technology might not yet have settled in 1972, when the data was gathered for the creation of the 217B model. Since Cm* uses mostly 1976-77 components, there are many 217B parameters that can be altered to take into account the maturity of the production process. Cm* data was compared with the predictions generated by various parameter changes in the model. As a result of these comparisons, a *modified* MIL 217B model was proposed: the quality factor for MOS chips was *derated* by a factor of 16. Based on this modified MIL 217B model, a PMS-level reliability model for Cm* was also presented [Siewiorek *et al.* 78b].

2.3.2. Transient Errors

Transient errors are manifestations of faults which are due to temporary environmental conditions. Very little is known about transient failures. Data collected on Cm* and other CMU computers has contributed to the understanding of this phenomenon. On Cm*, transient error data was collected by an Auto-Diagnostic program [Scelza 79]. The Auto-Diagnostic continuously exercised the Cm* system by running diagnostic programs on all otherwise idle computer modules. Whenever an error was detected by the diagnostic program, the information was printed on the console terminal. Looking through the console log, one could determine occurrences of transient errors. A more detailed analysis of Cm* error data was presented by Tsao [Tsao 78]. It was found that the interarrival times of transient errors follows a decreasing failure-rate Weibull distribution This is at variance with the standard assumption of constant failure rate (Poisson process, exponential distribution) used in reliability modeling. The Weibull distribution, observed on Cm*, was also observed on several PDP-10 systems [McConnel *et al.* 79a]. A summary of these findings will be presented in Section 2.3.3.

A set of four diagnostics are continuously run on the Cm's. These tests exercise (1) the memory, (2) the instruction set, (3) the traps and interrupts, and (4) the Slocal and a small part of the Kmap. The memory test is divided into 13 subtests, which include a gallop test, marching ones and zeros, and shifting ones. It takes approximately 13 minutes to complete one pass through 56K bytes of dynamic MOS RAM. The instruction-set test and the interrupt-and-trap test are designed to test the functioning the of LSI-11 processor. These are short tests, so many passes are done before moving to the next diagnostic. The Slocal diagnostic performs a number of functions. First it tests the registers and data path of the Slocal. Second, it exercises a part of the Kmap. Finally it runs a few tests on portions of memory.

Previously reported data [Siewiorek *et al.* 78a] indicated that several computer modules sometimes will report detection of errors almost simultaneously. Three basic occurrence patterns were noticed in transient errors: multiple errors occurring together in the same Cm (burst type), simultaneous errors reported by different Cm's (simultaneous), and finally, isolated errors (isolated). It was also observed that sometimes a single transient error event would be manifested as both the burst type and the simultaneous type together. Table 2-2 groups the recorded errors for the period between September 1977 and August 1978 into these four classes.

Table 2-2: Cm* Transient Error Events: Sorted by Mode and by Test Type

Type of Test	Memory	Instruction	Interrupt	Slocal	Total
Error Modes					
Burst only	8	6	1	16	31
Simultaneous only	6	1	0	20	27
Burst and Simultaneous together	3	1	0	10	14
Isolated	7	6	0	18	31
Total	24	14	1	64	103

Observations indicate that the most common cause of the burst errors is the destruction of the diagnostic program. A garbled diagnostic program can cause either spurious halts or a burst of reported errors, since successive restarts of the diagnostic program will be unsuccessful and result in consecutively reported errors. It is also possible that such burst errors arose when faulty transmission of code caused a bad copy of the diagnostic program to be loaded. But this is not likely because all such transfers are checksummed. Once a checksum error is detected, a reload is started. Bursts may also be caused by transient errors of a duration that is longer than the time resolution of the diagnostic. But the majority of observed burst type errors did not fit this hypothesis.

The simultaneous reporting of errors in several Cm's is the most interesting observation. It is

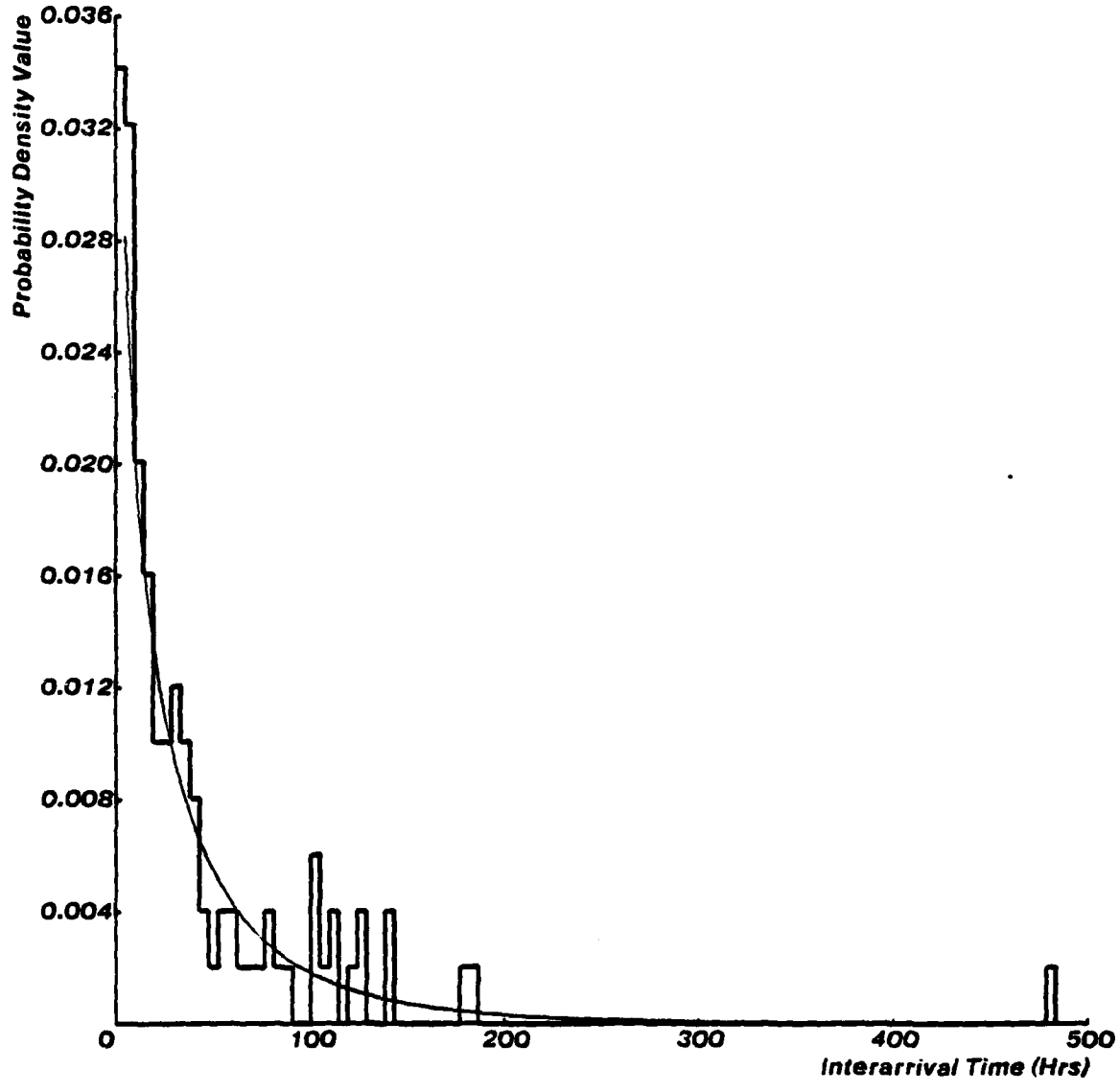


Figure 2-6: Distribution of Cm* Transient Errors, Sep. 77 to Aug. 78

conjectured that a systemwide transient failure causes this type of error. Two possible sources were proposed: Kmap error during Slocal test and common DC power supply glitches. It is known that turning power on and off in one Cm causes errors in other modules. These simultaneous errors could be human-induced events of this type that were not properly recorded in the system log. If these simultaneous errors were truly transient, one fourth (27 events) of all transient events affected more than one Cm.

2.3.3. Analysis of Transient Error Interarrival Time

Transient error data was processed and analyzed for the underlying statistical properties, with the aid of the SEADS transient-error statistical analysis program [McConnel *et al.* 79b]. Figure 2-6 shows the adjusted histogram of the interarrival for Cm* transient errors . The histogram of the distribution is overlaid with the maximum likelihood estimator (MLE) Weibull probability density function. Figure 2-7 shows the interarrival data for Cm* plotted on Weibull probability paper. The straight line drawn on the plot is a least-squared-error (LSE) linear fit to the data. Note that most of the visual deviation is due to relatively few points at the lower end. This deviation is mostly due to the transformation induced by the Weibull probability paper which is not very accurate for low-end data points. The near collinearity of the data points, tracking the LSE line, shows that the sample follows a Weibull distribution.

Table 2-3 lists some general statistics about the interarrival times for the five sets of data: TOPS-C system reloads on the CMU Computation Center DEC 2060 (under the TOPS20 operating system), PDP-10 system reloads on the Computer Science Department DEC KL-10 (under the TOPS10 operating system), PDP-10 memory-parity error interrupts on the KL-10, Cm* transient errors, and C.vmp[4] system crashes [McConnel *et al.* 79b]. In all cases, the mean is less than the standard deviation, indicating a decreasing failure rate ($\alpha < 1$). The Weibull shape parameter is α, and λ is the Weibull scale parameter.

For the last three sets of data, the 90% confidence intervals for α and λ were also generated. These values are listed in Table 2-4. Note that the range of values for α does not include 1.0, as it would have to, if the data did follow the exponential distribution.

Confidence-interval tests on the MLE Weibull parameters and Chi-square goodness-of-fit test confirmed the hypothesis that the data follows a decreasing failure-rate Weibull distribution. This is significant since past publications on the problem of transient errors have always assumed the exponential distribution for ease of computation. No other data have been published to support that

[4] C.vmp is a triplicated NMOS LSI-11 microprocessor with majority voting at the bus level [Siewiorek *et al.* 78b].

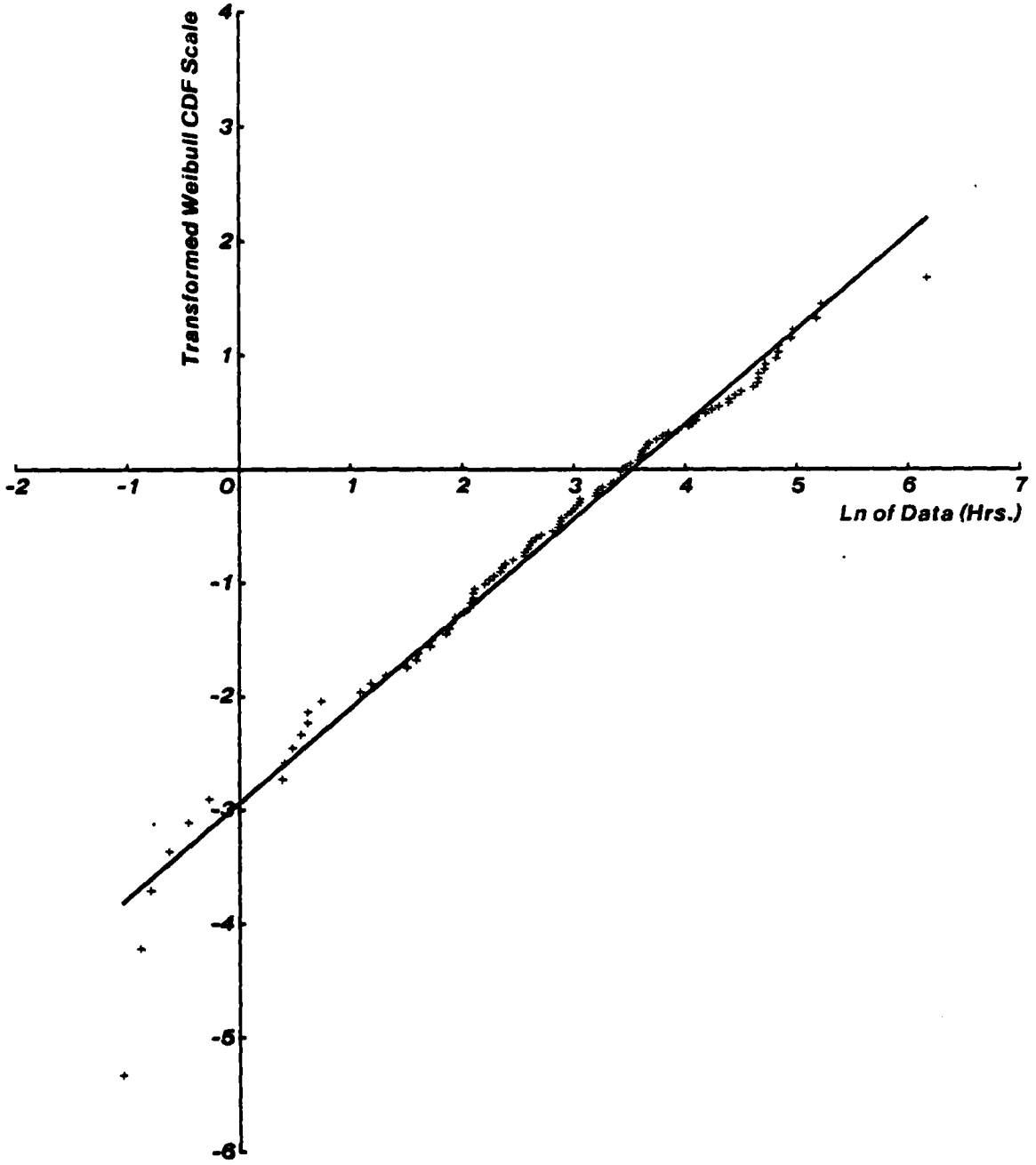


Figure 2-7: Weibull Plot of Cm* Transient Errors, Sep. 77 to Aug. 78

Table 2-3: Statistics for Transient Errors

	TOPS-C Reload	PDP-10 Reload	PDP-10 Parity	Cm*	C.vmp[5]
Time (hours)	2646	8576	8596	4222	4921
Errors	195	636	74	103	50
Interarrivals	196	640	78	104	51
μ(wall-clock time)	13.5	13.4	110.2	40.6	96.5 (328)
σ	16.5	24.6	244.9	59.8	167.8 (471)
α (Linear)	0.864	0.684	0.500	0.834	0.711
α' (MLE)	0.826	0.639	0.481	0.779	0.654
λ (Linear)	0.0843	0.109	0.0206	0.0294	0.0149
λ' (MLE)	0.0826	0.106	0.0203	0.0288	0.0146

Table 2-4: 90% Confidence Intervals for Weibull Alpha and Lambda

	PDP-10 Parity	Cm*	C.vmp
$[\alpha_{high},\lambda_{high}]$	[0.566,0.0307]	[0.893,0.0359]	[0.806,0.0214]
$[\alpha_{low},\lambda_{low}]$	[0.412,0.0134]	[0.693,0.0231]	[0.558,0.0099]

assumption. This observation of a decreasing failure-rate Weibull distribution means that the problem of modeling transient errors must be reconsidered.

2.3.4. Transient Error Data for February and March 1980

During February, 1980, the Auto-Diagnostic reported a total of 45 errors that were actual detected diagnostic faults found by individual test programs on the Cm's. It is evident from the console log that Cluster 3, Cm 14 had a hard failure, as 21 errors were reported. Errors reported simultaneously are assumed to be due to the same transient fault, and so only one transient error is counted. During this month, there were 4 simultaneous error events which caused the reporting of 5 redundant errors. After discounting these non-transient diagnostic errors, there were 19 transient-error events in February, 1980.

Table 2-5 presents the Mean Time Between Error (MTBE) for each cluster and the detected errors (sorted by tests). A new test, the parity test, is used on the 50 module Cm*. This test diagnoses the

[5] The pessimistic value discussed in [Siewiorek *et al.* 78b] is used throughout for C.vmp because there were too few interarrivals in the optimistic value (shown in parentheses for the mean and standard deviation) to be statistically significant.

parity generating and checking portion of each Cm. It is evident that the parity test is a very sensitive diagnostic test for transient errors, because many more parity errors were reported by this test than by the Slocal test which, in the past, used to report the largest single group of errors. The newly discovered sensitivity of the parity test is also reflected in the low MTBE, compared with a MTBE of 218 module-hours as reported in [Tsao 78].

Table 2-5: Cm* Transient Errors, February 1980

Cluster	1	2	3	4	5	Total
Module Hours	408	312	481	309	212	1722
Errors per Test						
Parity	1	3	7	2	3	16
Slocal	0	0	0	2	0	2
Instruction	1	0	0	0	0	1
Total Errors	2	3	7	4	3	19
MTBE (module hours)	204	104	69	77	71	91

Table 2-6: Cm* Transient Errors, March 1980

Cluster	1	2	3	4	5	Total
Module Hours	241	212	326	385	365	1529
Errors per Test						
Parity	1	2	3	1	3	10
Slocal	0	1	0	0	0	1
Instruction	0	1	0	0	0	1
Total Errors	1	4	3	1	3	12
MTBE (module hours)	241	53	109	385	126	127

During March, 1980, two more Cm's failed. Cluster 4, Cm 3 had a hard failure in the memory board. Also in Cluster 4, Cm 12 had a failure in the Slocal board. The transient error MTBE for that month is presented in Table 2-6. Data collection on Cm* will continue and analysis will be conducted periodically in the future.

2.4. References

[Bell 71] C. G. Bell and A. Newell.
Computer Structures: Readings and Examples.
McGraw-Hill, New York, 1971.

[Fuller *et al.* 77] S. H. Fuller, A. K. Jones, I. Durham, eds.
Cm review.*
Technical Report, Computer Science Department, Carnegie-Mellon University, June, 1977.

[Fuller *et al.* 78] S. H. Fuller, J. K. Ousterhout, L. Raskin, P. Rubinfeld, P. S. Sindhu, and R. J. Swan.
Multi-microprocessors: An overview and working example.
Proceedings of the IEEE 66(2):216 – 28, February, 1978.

[McConnel *et al.* 79a]
S. R. McConnel, D. P. Siewiorek, and M. M. Tsao.
The measurement and analysis of transient errors in digital computer systems.
In *Proceedings of the Ninth Annual International Symposium on Fault-Tolerant Computing*, pages 67 – 70. IEEE Computer Society, 1979.

[McConnel *et al.* 79b]
S. R. McConnel, D. P. Siewiorek, and M. M. Tsao.
Transient error data analysis.
Technical Report CMU-CS-79-121, Carnegie-Mellon University, May, 1979.

[Raskin 78] Levy Raskin.
Performance evaluation of multiple processor systems.
PhD thesis, Carnegie-Mellon University, August, 1978.
Available as CMU tech report CMU-CS-78-141.

[Scelza 79] D. A. Scelza.
An auto-diagnostic program for Cm*.
April, 1979.
CMU CSD internal report.

[Siewiorek *et al.* 78a]
D. P. Siewiorek, V. Kini, H. Mashburn, S. R. McConnel, and M. Tsao.
A case study of C.mmp, Cm*, and C.vmp: Part I—experiences with fault tolerance in multiprocessor systems.
Proceedings of the IEEE 66(10):1178 – 99, October, 1978.

[Siewiorek *et al.* 78b]
D. P. Siewiorek, V. Kini, R. Joobbani, and H. Bellis.
A case study of C.mmp, Cm*, and C.vmp: Part II—predicting and calibrating reliability of multiprocessor systems.
Proceedings of the IEEE 66(10):1200 – 20, October, 1978.

[Swan *et al.* 77a] R. J. Swan, S. H. Fuller, and D. P. Siewiorek.
Cm*—a modular, multi-microprocessor.
In *National Computer Conference, Proceedings, Vol. 46*, pages 637 – 44. AFIPS,

1977.

[Swan *et al.* 77b] R. J. Swan, A. Bechtolsheim, K. W. Lai, and J. K. Ousterhout.
The implementation of the Cm* multi-microprocessor.
In *National Computer Conference, Proceedings, Vol. 46*, pages 645 – 55. AFIPS, 1977.

[Swan 78] R. J. Swan.
*The switching structure and addressing architecture of an extensible multiprocessor, Cm**.
PhD thesis, Carnegie-Mellon University, August, 1978.

[Tsao 78] M. M. Tsao.
A study of transient errors on Cm*.
Master's thesis, Carnegie-MellonUniversity, December, 1978.

3. Exploiting the Hardware: Basic Software Support

An important part of any systems-development project are the tools which are used to realize it. It is rarely possible, however, to duplicate the utility programs of a mature operating system on an experimental machine. Rather than spend our time on the routine tasks of developing editors and file systems and transporting compilers, we have chosen to use these utilities on the Computer Science Department's PDP-10's. This has enabled us to concentrate our efforts on implementing translators for Kmap microcode, and debuggers for microcode and software. We have also built an efficient mechanism for transporting files to Cm*. This chapter describes these tools.

3.1. The Cm* Host

Gregg Lebovitz

When software was first being developed on Cm*, there arose a need for a system to let the software developer control the usage of resources on the machine, and protect his resources from being disturbed by other users. The **Cm* Host** system was developed for this reason. The Host is a serial-line oriented resource-management facility running on a PDP-11/10 with 28K words of memory.

The command structure of the Host resembles that of the PDP-10, which most Cm* users are familiar with. The Host performs several of the functions of a primitive operating system.

- *Security.* The Host protects Cm* from unauthorized use by providing an account system. In order to use any Cm* resource, one must *log in* to the system by typing in a valid user number and password.

- *Protection.* A resource on Cm* must be assigned to a user before it may be used. Once it is assigned, no other user may access the resource until it is deassigned. The resources of Cm* are thus partitioned among users. Development work on several projects, even several different operating systems, may be carried out simultaneously using disjoint sets of resources (different clusters, for example). Meanwhile, each user is protected from having his working environment accidentally disturbed by another user.

- *Resource control.* The Host has commands for controlling a variety of assigned resources. Among these are commands to load programs from tape or from the PDP-10; to start, halt or single-step a processor; to guard against buffer overflow; and to communicate with a resource via a direct terminal link.

- *Communication.* The Host allows the user to monitor output from all assigned resources simultaneously, by requesting that messages sent to him by each resource be printed at his terminal. As an option, a **banner**, that is, an abbreviation of the name of the resource, may be printed in front of each message to indicate where the terminal output came from.

> Such a feature helps the user to monitor the interaction of resources for debugging purposes. A user may also send a message to any serial line from the Cm* Host. This facility enables the user to communicate simultaneously with several of his processes—or with other users. This facility is used by the Cm* Auto Diagnostic system to notify the diagnostic processor of errors detected on other resources.

Four different kinds of serial lines are available to the user. These are lines connected to Cm's, Hooks processors, terminals, and the Computer Science Department's Front End computer. The Cm lines are configured two per cluster. These lines allow the user to load code and data directly into a few of the LSI-11's, and then to interact with with his running programs. The Front-End lines have two purposes. They allow the user to communicate with other computers from Cm*. Many programs on Cm*, for example, the DA Link programs (Section 3.6) interact with programs on other machines. The Front-End lines also permit the user to monitor debugging information on both machines simultaneously from one terminal using the banner facility mentioned above. The lines to the Hooks processors allow the user to load Kmap microcode and to communicate with the Kmap debugger (Section 3.4).

The original Host system was written in the fall of 1975 by Hal Van Zoeren. Don Scelza and John Ousterhout completely rewrote it with enhancements in early 1977. The Host has been running smoothly since the summer of 1977. Its average uptime is several hundred hours. The features it provides have greatly facilitated the implementation and debugging of software on Cm*.

3.2. The CMIC Microassembler and its Software Support

Ed Gehringer and Steve Vegdahl

The CMIC microassembler has been used to produce most of the microcode which has been written for the Kmaps. It was adapted by Paul Rubinfeld from an earlier microassembler in the fall of 1976, and consists of 2100 lines of source code. Writing without macros, the programmer codes each separate micro-operation by specifying a numeric value for each non-zero field of the microinstruction. Fields which are not assigned a value are set to zero. Statements have the form `<fieldname> = <value>`. All of the statements on a single line are assembled as part of the same microinstruction. Continuation lines are allowed. A microinstruction, then, might be written something like this:

```
fcn = 3 ; reg = 13 ; rw = 1 ; na = 20
```

The programmer does not normally write code in this fashion, however, as the macro facility allows him to define macros which expand to a set of closely related micro-operations. Thus the example above, which causes addition to be performed by the ALU (`fcn = 3`) and stores the result in general-purpose register 13 (`reg = 13 ; rw = 1`), and then branches to location 20 of the control store (`na`

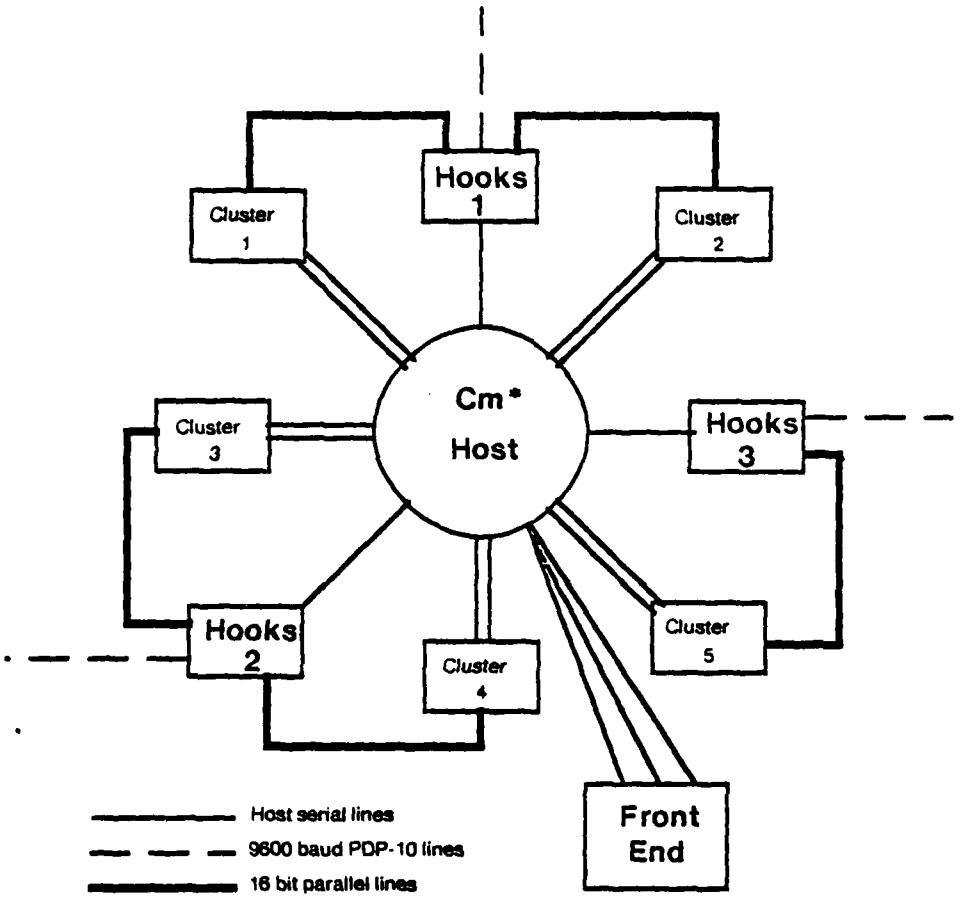


Figure 3-1: Lines Connected to the Cm* Host

* 20), might be programmed as follows with suitably defined macros:

```
Add&ld(gpr(13)) ; goto(ErrorBranch)
```

Here the first macro invocation expands to the first three micro-operations, and the second macro invocation generates the branch operation. `ErrorBranch` is assumed to be a label attached to location 20. In practice, there are a few hundred macros which are used by all projects. Additional project-specific macros are routinely defined. Deeply nested macro calls are not uncommon.

The other major function performed by CMIC is the placement of instructions within the control store. Placement is complicated by the way the Kmap does conditional branching. The Kmap has no program counter; rather each instruction contains a *next-address field*, and conditional branching is performed by oring bits into this field. This constrains certain microinstructions to be placed only in

certain locations. For example, if a sixteen-way branch is to be performed by oring a four-bit value into the low-order four bits of the next-address field, the sixteen microinstructions which can be the object of this branch must be placed in sixteen consecutive locations, beginning with an address divisible by 16. The existence of several different kinds of conditional branches means that CMIC must often rearrange the microinstructions considerably in order to fit the microcode into the smallest possible block of control store.

CMIC provides only the bare facilities of a microassembler. It provides no linking mechanism to resolve external references in separately assembled microcode files. Nor does it aid in register allocation across subroutine calls. The PMIC program, consisting of 2000 lines of source code written by Pradeep Sindhu in the summer of 1978, provides these facilities. It binds subroutine names to absolute addresses, so that they can be referenced symbolically by routines in other microcode source files.

For efficiency reasons, it is not practical to save registers across microcode subroutine calls. Yet the registers used by a particular micro-routine must not be disturbed by any micro-routine which may be reached from it by a series of subroutine calls. PMIC performs register management by allocating the Kmap's 32-general purpose registers among subroutines.

When the programmer writes microcode, he uses macro names in place of absolute register numbers. PMIC reads the microcode, analyzes the subroutine call graph, and writes the macro definitions for the symbolic register names used by the programmer; that is, it decides which absolute register number to assign to each macro name. Then, when the programmer assembles his code with CMIC, he passes the macro definitions which bind symbolic names to register numbers.

From a microcoding point of view, it would have been simpler to implement the Kmap's registers as a stack. Not only would this have obviated the inconvenience of running PMIC, but it would also have allowed recursive subroutines. This option was considered when the Kmap hardware was designed, but it was rejected because there was not enough room on the processor board for the additional chips which would have been needed.

3.3. The MUMBLE Microcode Compiler *James Gosling*

MUMBLE, a most unlikely microassembler, is a compiler, written by James Gosling, that generates high-quality microcode. Its design was prompted by the need for better tools to assist in the production of microcode for the Kmaps (Section 2.2). These are horizontally microprogrammed machines with many complex data paths and timing rules. Using conventional techniques for microprogramming, operations that are logically related have to be physically separated to satisfy

timing constraints and merged with unrelated operations in order to exploit parallelism and produce compact code. The major goals of MUMBLE are to allow logically connected primitive machine operations to be written together with little or no loss of efficiency, and to provide high-level-language control-flow constructs.

When MUMBLE was designed, no attempt was made to hide the gross aspects of the structure of the micromachine—its registers, busses and communication with the outside world. Rather, MUMBLE was designed to hide fine details like inter-instruction timing and address assignment. This approach was based on the premise that to hide all of the structure of a machine with a high-level language would lead to unacceptable performance. Most micromachines have unusual features whose exploitation is crucial to effective use of the machine.

At the most primitive level, MUMBLE programs are written in terms of micro-operations. Programmers can actually specify which micro-operations that they want performed, although they have no control over placement. In practice though, micro-operations are almost never explicitly used, "paths" and macros are usually used to build more powerful and more machine-independent abstractions.

MUMBLE also attempts to make control flow within a program more obvious by providing "high-level language" features such as **if-then-else-fi**, **repeat-until**, **case-of-end** and procedure calls with parameters. With the present poor interblock optimizations these statements will often cause the generation of suboptimal code, but MUMBLE provides escape clauses so that it can be coerced into producing better code. The use of such techniques is strongly discouraged since they tend to obscure programs. The current MUMBLE compiler takes as input a MUMBLE program and translates it into a program for a microassembler. Currently CMIC, the microassembler for the Kmaps, is used.

Most of the methods which the compiler uses are applicable to many horizontally microcoded machines. Almost all of the compiler's knowledge of the micromachine is provided by a file of definitions which could be replaced by one describing another machine. This definition file contains information about what micro-operations the machine can perform and how the micro-operations are related in resource usage and timing. Using these definitions, the compiler produces microcode that takes a near-minimum number of microwords.

The optimizations performed by the compiler are limited to a fairly intelligent packing algorithm (which handles unstable resources), limited interblock code motions, and a minimum-path-length data routing algorithm.

Micro-operations. The basic units out of which MUMBLE programs are constructed are **micro-operations** which specify primitive operations performable in a single instruction time by the micromachine. For example, in the Kmap definition there is a micro-operation "fbus.plus" which represents the single ALU operation of adding the A register to the B register and putting the output on the F bus. All timing/resource dependencies in the micromachine are defined at the micro-operation level. Dependencies are expressed in terms of **classes**, which are sets of micro-operations; each micro-operation may belong to several of these. Each class has specific resource-usage or dependence properties. For example the Kmap micro-operation that loads the data-RAM address register from the A bus is defined to be a member of the class of micro-operations that use the A bus value from the previous microcycle; there is also a class of micro-operations that set the A bus. A relationship is defined between these two classes; namely that the micro-operation that uses the value on the A bus must be in the microword following the micro-operation that sets the A bus.

Paths. Many operations that one wishes to perform on a micromachine involve only the movement of data from one place to another. Unfortunately, many conceptually simple data movements can become quite complex when microprogramming: for example, data must be explicitly shuffled from a register or bus, across other busses and through intermediate registers until it finally arrives at its destination. The micromachine definition file also describes how the various parts of the micromachine are interconnected and which micro-operations are used to move data between connected points. The user can then write statements of the form $A \rightarrow B$, where A and B are registers or busses in the micromachine, and the compiler will find the route through the least number of intermediate nodes in the data-path graph, and will emit a sequence of micro-operations that correspond to that path. These interrelations are expressed in a graph whose nodes are the registers and busses of the micromachine and whose edges are their interconnections.

Paths may have intermediate stages. For example $A \rightarrow B \rightarrow C$ is equivalent to $A \rightarrow B$; $B \rightarrow C$. These intermediate stages may act as filters, effecting a transformation as data passes through them. In the Kmap definition the node *IncOne*, which doesn't correspond to any single part of the real machine, may be used as a filter that adds one: $A \rightarrow IncOne \rightarrow A$ adds one to A.

Macros. A macro facility is provided in MUMBLE to allow more powerful abstractions to be built. For example, it is possible to define a macro called *plus* so that the statement *plus(r1,r2)→r3* adds *r1* to *r2* and places the result in *r3*. Macros may be called recursively: *plus(plus(r1,r2),r3)→r4* does the obvious thing; and macros may define other macros.

High-level statements. Linear blocks of code composed of micro-operations which are explicitly written or result from the expansion of a macro or a path are put together in a framework of high-level statements. These statements provide control structures that are largely independent of the actual *control mechanisms* provided by the micromachine.

The lessons of structured programming teach us that we want to use higher-level control constructs. This is difficult on micromachines since conditional branches are often effected by oring values with the next address field of an instruction. Using conventional techniques this leads to the confusing appearance of "magic" numbers throughout a program. The control constructs of MUMBLE were designed to hide this from programmers.

Often, just a single bit is to be ored with the address of the next instruction. Such single bits can be thought of as boolean variables which MUMBLE allows the programmer to use in the test portion of **if-then-else-fi** or **repeat-until** statements. For example, in the Kmap it is possible to **or** the low-order bit of the F bus into the next instruction address; in MUMBLE's definition of Kmap the boolean *f.is.odd* is defined so that the statement "**if** *f.is.odd* **then** ... **else** ... **fi**" would have the natural meaning. Where more than a single bit is being ored the **case** statement can be used. There is also, of course, a **goto** statement. It is interesting to note that **goto** was the most difficult statement to implement since the analysis of interblock timing had to take into account widespread interactions.

Procedures. The procedure facilities in MUMBLE allow calling sequences to be specified on a procedure-by-procedure basis. This is done to allow the efficient exploitation of the various registers and busses of the machine when passing parameters. Parameter passing and result return may thus involve timing dependencies that cross entry/exit boundaries. This is complicated even further by the fact that a procedure call and a procedure body may be separately compiled. For example, one may want to call a procedure and pass a parameter on a bus that is stable for a short period of time. The parameter must be placed on the bus immediately before the call and it must be used immediately upon entry.

MUMBLE requires that the specification and the implementation of a procedure be separated. The specification gives all information necessary for separate compilation. The necessary timing information is not given explicitly; rather the user must provide in the specification a sample of the code that will surround the entry and exit of the procedure, and the timing information is deduced from this. In order to ensure that this sample is given correctly, deduced information can be compared with the actual timing that surrounds procedure calls and bodies when they are compiled. This actual information from the bodies and calls is used only for checking since timing information from the body may influence (and be influenced by) the call, which may be separately compiled.

The operation of the compiler. The micro-operations for each linear block are formed into a graph with timing and data dependencies linking them together. These graphs are then transformed into a linear form, assigning micro-operations to microwords, using a packing algorithm derived loosely from the topological sorting algorithm given in [Knuth 73]. Experiments have shown that this algorithm for packing micro-operations into a linear block almost always does as well as an experienced programmer. Some limited interblock code motion is performed, but not much work has been done in this direction. The major limit on the quality of the code produced by the compiler is this lack of interblock code motion.

As an example of what the packing algorithm does, consider the *plus* macro from section 3.3.0, which actually appears in the Kmap definition file. It can be used in a statement like *plus(fab,fbl)→dadr* to compute the sum of the *fab* and *fbl* registers and place it in the *dadr*. This statement generates code which occupies two microwords of storage. If the statement following *plus* were *minus(r1,k)→r2*, which also occupies two microwords of storage, then the two statements together would only occupy three microwords since MUMBLE would merge them.

A Tiny Program. The following fragment is the inner loop of a block-transfer procedure.

DestinationAddress, **SourceAddress** and **WordCount** have been bound to registers; **ones** is a fictitious register[6] that behaves as though it contains the constant −1; **read** is a procedure that takes an address as an argument and returns the value of the word in the memory of the current master Cm at that address; **write** is a procedure that writes a word at a specified address; F is the F bus; and **F.eq.ones** is *true* if the value on the F bus is −1.

```
repeat
    write(DestinationAddress,read(SourceAddress));
    SourceAddress→inctwo→SourceAddress;
    DestinationAddress→inctwo→DestinationAddress;
    plus(WordCount,ones)→F→WordCount;
until F.eq.ones;
```

The code generated by the compiler for this example is very good.

Users' Experience. The main use of MUMBLE has been in the ECHOES experiment (Chapter 9) by Mike Kazar, who has been quite enthusiastic about the compiler. The other two operating system groups did not use MUMBLE because the compiler was not in a usable state when they needed to begin coding. From small experimental programs and Mike Kazar's experience, MUMBLE appears to

[6] *Ones* is actually a MUX selection on one leg of the ALU, hence efficient to generate. It's a deficiency in the compiler that the user has to know that there are special mechanisms for generating the constants 0 and −1.

represent a useful approach to microprogramming. The micromachine specification technique appears to be quite good; very few errors in the original specification of Kmap were found, and those that did occur were easy to detect and correct. Much work remains to be done, particularly in the areas of interblock code optimization and testing the compiler on other machines.

3.4. KDP: A Flexible Debugging Environment *Anita Jones*

The **Kmap Debugging Package** (KDP) assists in diagnosing the Kmap hardware and debuggingthe Kmap microcode from a remote terminal. KDP is a BLISS-11 program that executes on a Hooks processor. Using the Hooks interface to a Kmap, a user may start and stop the clocks within the Kmap, examine and change internal Kmap state, as well as load, trace and symbolically edit microcode.

KDP is capable of interacting with up to 8 Hooks-Kmap interface units, though it communicates with only a single "current" unit at a time. In the current Cm* configuration, each of the three Hooks processors is connected to one or two Kmaps. To initiate a debugging or diagnostic session, the user requests KDP to load a microcode object file into the current Kmap via a 9600-baud Hooks-to-PDP10 line. The user may initiate Kmap execution of the object code and proceed with diagnosis or debugging.

KDP commands are generally non-destructive. They may be invoked at any time without disturbing the state of the Kmap. If the Kmap is running, KDP will automatically stop the clock, perform the command and restart the clock (if appropriate) so that there will be no effect on the Kmap, except that caused by the command. For example, consider a user attempting to debug an operating system operation that involves two clusters, both of whose Kmaps interface to the same Hooks processor. If the user desires to interrogate or trace the actions of the Kmap in the cluster originating a request that is destined to be satisfied by the second Kmap, he may set breakpoints or perform tracing at the origin Kmap, leaving the remote Kmap to run without hindrance.

KDP operations may be divided into several groups. We describe each group only to the ext nt of giving a flavor of the operations available. Complete documentation is available in [Ousterhout 78].

- *Initialization and clock control.* Either the Pmap clock or the Kbus clock may be stopped and single-cycled an arbitrary number of times. The user may request tracing so that the addresses of the instructions being executed are printed out. KDP can be used to reinitialize the Kmap hardware, and to restart execution at an arbitrary address.

- *Memory control.* Besides allowing the user to load the control memory by naming a PDP-10 file, KDP will display and change both the control memory in which code is stored and the memory used strictly for data (data RAM). The user may either display the contents of

the control memory in octal, or name an instruction field symbolically and then change or display only that particular field.

- *Execution monitoring.* KDP permits up to eight breakpoints to be set simultaneously for a single Kmap. In addition, the user may request that a set of Kmap state values are to be dumped whenever the Kmap halts a specified number of times. After halts, KDP will automatically restart the Kmap clock if so requested. For Pmap diagnosis, KDP will stop the Kmap clock after certain parity errors are encountered, and will cause alternative values to be gated into certain registers for test purposes.

- *Interrogation.* KDP will display a large number of register values: the 32 general-purpose registers associated with the currently executing context, the 32 subroutine registers, and the 35 additional Pmap internal state values that are accessible to the Hooks interface unit. This latter set includes most register and bus values involved in the ALU usage, computer-module memory access, linc-bus access, Pmap program-counter control, and control-store and data-RAM access.

Assay of KDP. KDP was developed by John Ousterhout, primarily in the summer and fall of 1976. A total rewrite in the spring of 1977 cleaned up its structure without substantially changing its functionality. Since then members of the Cm* project have developed substantial amounts of microcode. Together the SMAP, STAROS, MEDUSA, and ECHOES microcodes, described later in this report, comprise nearly 100,000 lines of code, including comments, and approximately 11,000 microinstructions.

We do not believe that this amount of microcode could have been successfully developed without the aid of the Hooks interface to the Kmaps and the software packages that make the Hooks interface tractable. These tools allow us to suspend Kmap execution arbitrarily, to observe progressive state values during execution, and to adapt Kmap state without going through the cycle of editing microcode source, recompiling it, reloading the Kmaps, and executing the altered code. It is our experience that such a cycle costs roughly half an hour to over an hour at prime compute times.

In addition, the Hooks interface permits execution of a variety of Kmap diagnostic programs not described here. Maintenance of the Kmap has been made decidedly easier. As a result Kmap availability is extremely high, an average of one failure every six months or so. The major factor in reliability is probably the very conservative hardware design, although the Hooks interfaces have helped to reduce mean-time-to-repair (the diagnostics usually indicate within 5 chips where the bad one is).

3.5. The StarOS Test-Bed

Ivor Durham

The StarOS Test-Bed, written by Ivor Durham, is a tool for developing software components of a Cm* operating system independently of each other. Components are tested and debugged in a hospitable uniprocessor PDP-11 environment provided by the Test-Bed. It is known as the StarOS Test-Bed because it builds data structures used in the StarOS operating system, though with relatively minor changes, it could be used with other operating systems as well.

The Test-Bed has been used to considerable advantage in several ways. First, after a few initial StarOS components had been debugged and included in the Test-Bed, other StarOS components were developed independently using the augmented Test-Bed. Second, because the interface to the various StarOS functions is well defined in the design documentation, independent testing of software modules reduced the number of problems encountered when components were put together. When a problem was encountered in an integrated system, the programmer could retreat to a Test-Bed environment to locate the problem within an individual component. Finally, we believe that by removing many of the initial bugs in a uniprocessor environment the problems encountered in the multiprocessor environment will be primarily those of interaction (sharing and synchronization).

The Test-Bed itself is a collection of software modules—a few of which are specific to StarOS—that may be linked together with "user" software to produce a monolithic PDP-11 program. These components include I/O functions, a command interpreter, a library of functions for constructing interactive interfaces to software under test, and a facility for reporting PDP-11 detected errors. All of the facilities of the StarOS microcode are available through Test-Bed commands. Also provided are functions to interpret capabilities in a readable form and functions for tracing simple subroutine calls. Various simple debuggers are available (like DDT), but none of them embodies any knowledge of the StarOS object world and so they are of limited use. A version of the main debugger that was used on the C.mmp multiprocessor (known as SIX12) has been adapted for the StarOS object environment and it will ultimately provide facilities for debugging multiple processes via a common interface in user space.

A typical stratagem for testing a StarOS component with the Test-Bed is to construct a user-interface function from convenient library components and include the interface function in the Test-Bed as a new command: this is achieved simply by adding the function name to a private copy of the Test-Bed command table and re-compiling the table. Software modules are linked together with the Test-Bed components and the entire program executed on a Cm* LSI-11. Parameters may then be specified and functions invoked via the Test-Bed's interactive command interface.

As functions are invoked, any errors reported by the LSI-11 processor and those reported via a

Kmap interrupt are interpreted for the user, unless functions have been provided to handle these exceptions unaided by the Test-Bed. In addition, any exception that is handled by the Test-Bed also causes a code checksum to be verified so that the user is warned if any damage has been done to the Test-Bed or functions under test as a cause or effect of the exception.

3.6. The Cm* File-Transfer System

Gregg Lebovitz

The systems software being developed for Cm* is written, compiled and stored on a remote machine. Consequently, object code must frequently be transferred to Cm*. Until recently all files were transferred by way of a 1200-baud serial terminal line. Such transfers are quite slow and prone to transmission errors. Transfers of large files to Cm* can, under unfavorable conditions, take as much as 15 minutes. For this reason, the DA Link file-transfer system was developed. The DA Link is a 10-megabaud bit-parallel DMA link between Cm* and a DEC-10; the transfer protocols, of course, reduce the effective throughput. It is used to transfer LSI-11 object code from the DEC-10, where it is developed, to Cm*, where it is to be executed The file-transfer system is a set of software that resides on the DEC-10 and implements reliable file transfers.

The hardware for the DA Link consists of a DMA device, called a DA28-C, that interfaces several parallel lines to the memory I/O bus on the DEC KL10; a home-built DMA device that interfaces a parallel line to the Unibus on one of Cm*'s LSI-11s; and a sixteen-bit parallel line.

The file-transfer system is composed of two parts. The first is a set of low-level reliable transfer routines. These routines are designed to provide an error-free transmission of packets of data between Cm* and the DEC-10. The higher-level file-transfer routines interface the reliable packet-transfer software to the DEC-10 file system. The file-transfer routines can also multiplex data from several files, allowing a single physical link to become a multi-user system.

Reliable Transfer Routines. The purpose of the **reliable transfer routines** is to transfer data packets across the DA Link. These routines implement the **reliable transfer protocols**, which provide a convention for transferring the data without uncorrected errors.

The reliable transfer routines transfer all data across the DA link in packet format. A packet consists of two sections, the packet header and the packet data. The packet header contains four sixteen-bit words of information. The first word holds the packet sequence number. Its purpose is two-fold: to identify a packet uniquely, and to specify how many packets were sent before this packet. This is necessary for the implementation of error recovery, as we shall explain later. The second word of the packet header is the packet function code, which determines how the data in the packet is to be

interpreted. The third word contains an integer between 0 and 252 that specifies the number of bytes in the packet data section. The fourth word is the checksum of the packet. The remainder of the packet holds the packet data.

The packet sequence number identifies a packet and ensures that it was received in sequence. Originally, the sequence number was to be used to implement a system in which packets could be sent and received out of order: all requested data would have been sent without any communication of transmission errors from the receiver to the sender. When all data had been transferred the receiver would have re-requested all packets that were received with errors. This would have eliminated the need for timely acknowledgment protocols. This system was not implemented since hardware transmission errors proved to be too infrequent to warrant the time and effort to implement such software. The present system does not allow the receiver to request individual packets. Instead it must request all the data that was sent after the point where the first error occurred.

The file-transfer system employs a **packet quota system** to prevent the receiving site from overflowing its buffer. The quota system establishes the size of the receiving site's buffer at the time a connection is made between the sender and receiver.

The second word of the packet header specifies that the packet will perform one of five functions: the **Connect** function is used to establish a connection between two machines on the DA Link. It serves to synchronize the two machines on the DA Link and set up the buffer quota system. The **Disconnect** function tells the other machine on the DA Link that the first machine no longer wishes to communicate across the link. The **Data** packet is used to transmit data across the DA Link.

The **Acknowledgment** packet serves two purposes. It carries both packet-acknowledgment and packet-quota information. Three words of data are associated with it. The first word is the sequence number of the first packet received with errors, if an error has occurred. The next word tells whether there was an error in the previous transmission. The last word is the new quota information.

File Transfer Routines. The **file-transfer routines** perform the functions of opening and closing files, reading and writing data to and from files, and moving file pointers. To provide a multi-user environment for the DA Link, the file-transfer system implements **virtual channels**, which are bi-directional communication paths over which control data and information may pass. The file-transfer system allocates a buffer for each virtual channel and provides buffer destination information for the data. A file is associated with each virtual channel, and all data placed in the virtual channel's buffer is transferred to this file.

Information handled by the file transfer routines is transferred to and from the reliable transfer routines in the form of packets. These packets are much like the packets used by the reliable transfer

routines themselves. Each packet contains packet-header and packet-data sections. The packet header contains the virtual channel name and the function the packet performs. The packet-data section contains packet data and has no size restrictions.

Status. The file-transfer system has been running smoothly since January 1980. The DA Link is a great advantage over the 1200-baud serial terminal line. Its effective throughput, on top of all the protocols, is 10 to 40 kilobaud, depending on the load on the DEC-10.

Presently the file-transfer system is running as a daemon program on the DEC-10 and can only be accessed from Cm*. Future versions will use the DEC-10 interprocess communication facility and will allow the daemon program to be accessed by users logged in on the DEC-10. Eventually a "telnet" facility will be added to allow users logged in on one of the machines to attach their terminal to a job on the foreign machine via the DA Link.

Don Scelza designed the DA Link protocols. Scelza, John Ousterhout, and Gregg Lebovitz implemented the user interface on the DEC-10. John Ousterhout and Howard Wactlar wrote the additional DEC-10 monitor code which was required to support the DA28-C; and Ousterhout wrote the PDP-11 code.

3.7. References

[Knuth 73] Knuth, Donald E.
The Art of Computer Programming, Volume I: Fundamental Algorithms.
Addison-Wesley, Reading, MA, 2nd edition, 1973.

[Ousterhout 78] J. Ousterhout.
Cm Kmap microprogramming manual and debugger manual.*
Technical Report, Carnegie-Mellon University, July, 1978.

4. Issues in the Design of Cm* Microcode

Standing as the switching center within a cluster of computer modules and as a node on a network of clusters, the **Kmap** is the primary source of communication and synchronization in the Cm* system. For a hardware description of the Kmap, see Section 2.2. Although we frequently refer to the Kmap as the microprogrammed entity, the Cm* systems microprogrammer actually writes code for the **Pmap**, which is only one component of the Kmap. Taking the viewpoint of the Pmap microprogrammer, this chapter discusses the operation of the Kmap, describes the interfaces between Kmap and computer module and between Kmap and Kmap, and presents some examples of the standard problems of distributed systems which arise at the Kmap microprogramming level. The chapter concludes with a description of an early version of Kmap microcode whose design ignores most of these problems.

4.1. The Kmap as a Transaction Controller *Thomas Rodeheffer*

In the Cm* system, Kmaps and computer modules communicate with each other by sending message packets along some interconnecting communication bus. Although a computer module can only send and receive a few types of message packets—packets dealing with memory accesses which are handled automatically by the Slocal—the Kmap can be programmed to act in almost any manner desired by the microprogrammer. This programmability is one of the strengths of the Cm* *intercommunication structure.*

The Kmap is envisioned as a transaction controller, sending to and receiving from computer modules and other Kmaps message packets which contain requests and replies, following a protocol designed by the microprogrammer. Because one of the common transactions that the Kmap is expected to handle is the mapping of a memory access issued by the processor of one computer module to a location in the memory of another computer module—a transaction which must be performed very rapidly and with little delay if any reasonable system performance is to be obtained—the Kmap contains some special hardware features designed to assist it in controlling many transactions at a high rate of speed.

The Kmap hardware supports eight separate Pmap **contexts**, each with its own set of general purpose registers and its own micro-subroutine stack. Typically, each context is in charge of one transaction. When one context needs to wait for a message packet to return with the reply to some lower-level request, it has the Pmap switch to another context so that work on some other transaction

can proceed concurrently.

The Kmap also contains 5120 words of random-access memory, called the **data RAM**, which the Pmap can read and write at the expense of a few microinstructions. Because the data RAM is shared by all contexts, it typically holds information which is of interest to more than one transaction, such as cached pieces of address translation tables and mechanisms for synchronizing the use of other resources among different contexts.

Taken individually, each Kmap appears to the Pmap microprogrammer as a non-preemptive, hardware-scheduled multiprogramming system. Taken collectively, the network of all Kmaps presents the microprogrammer with a distributed system with message-packet intercommunication. As the examples in Section 4.4 illustrate, the problems of programming this system are not trivial.

4.2. The Interface between Kmap and Computer Module

The Pmap communicates with the computer modules in its cluster via the **map bus**, a packet-switched bus controlled by the **Kbus**. The Kbus fields requests and replies from computer modules, coordinates the transfer of data across the map bus between computer modules or between a computer module and a Pmap context, and keeps track of which Pmap contexts are free to service new requests. Two queues, the Kbus out queue and the Pmap run queue, provide the interface between the Kbus and the Pmap. Refer to Figure 2-4.

Whenever the processor of a computer module issues a non-local memory access (see Section 2.2), a **service request** is signalled to the Kbus. The Kbus allocates a Pmap context, reads via the map bus the virtual address for the memory access, and activates the new context by placing an entry in the Pmap **run queue**. The computer module which thus invokes a Pmap context to process its memory access is called the **master computer module** or **master Cm**.

The Kbus activates (or reactivates) a Pmap context by placing into the Pmap run queue an entry containing the number of the Pmap context to be activated along with a small amount of other data, such as the virtual address of the processor's non-local memory access or the result data from a Pmap-initiated memory access. A newly-activated context does not automatically gain control of the Pmap, however, but instead must wait on the run queue until the Pmap removes its entry and loads the new context number. Since the Pmap examines the run queue only under the explicit direction of the current Pmap context, a context cannot unwillingly lose control of the Pmap. Loading a new context number from the run queue performs what is called a **context swap**: a context which invokes a context swap is said to **swap out**.

A Pmap context can initiate a physical memory access in any of the computer modules in its cluster. This memory access can either be a read access, in which case the result is the data read, or a write access, in which case the Pmap also supplies the data to be written and the result serves merely as an acknowledgement. The context invokes the proper Kbus operation and the Kbus sends a memory access request via the map bus to the indicated computer module, called the **destination computer module** or **destination Cm**. When the destination Cm completes the memory access it signals a **return request** to the Kbus, which reads the result of the memory access via the map bus and reactivates the requesting context.

The Pmap invokes a Kbus operation by loading a request into the Kbus **out queue**. About two dozen different operations are provided, most of which are variations of the Pmap-initiated memory access. After loading a request in the out queue, the Pmap context typically swaps out in order to let work on some other transaction proceed concurrently with the operation requested of the Kbus. Later, when the result of the operation becomes available, the Kbus reactivates the requesting Pmap context. This sequence of requesting a Kbus operation, swapping out to run other contexts, and waiting for eventual reactivation is an extremely common occurrence in Pmap microcode.

4.2.1. A Simple Kmap Operation

From the point of view of the computer module, a non-local memory access causes the invocation of some **Kmap operation**, the result of which is reflected back as the result of the memory access. The most basic Kmap operation, and certainly one which is expected to be invoked frequently, is the mapping of the non-local access to a location in the physical memory of some computer module in the cluster. Because it is so common, a mapped non-local memory access is usually just called a **mapped reference**. See Figure 4-1. Each memory access issued by the processor of a computer module passes through its Slocal, which either routes the access directly to local memory or sends it out to the Kmap. A memory access which is handled by the Kmap can be mapped back to the local memory of the issuing processor, but a direct local access is about three times faster.

The simple, intracluster, mapped memory access proceeds as follows. The master Cm signals the Kbus, which activates a Pmap context. The Pmap context considers the virtual address, the master Cm number, the contents of an address-translation table, and any other desired source of information, and determines the destination computer module and the physical address within that module to which the memory access is directed. A Pmap-initiated memory access is performed on the desired memory location and the result returned to the master Cm.

In order to obtain better performance, the Pmap microcode can actually use a special case of the Pmap-initiated memory access: one in which any data to be written is supplied by the master Cm

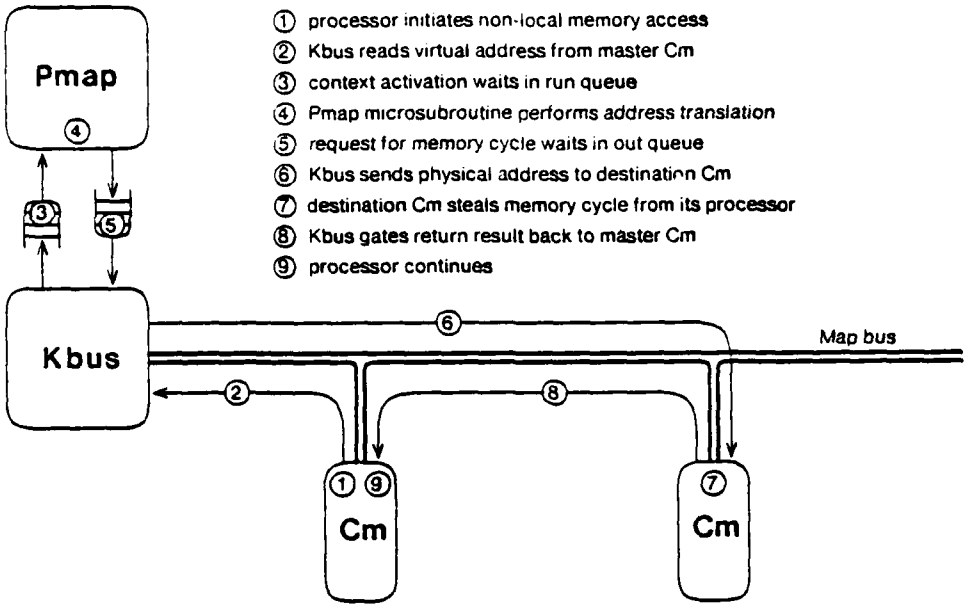


Figure 4-1: The Steps in an Intracluster Memory Access

instead of the Pmap context and, provided that no error is indicated, the result data from the destination Cm is routed directly back to the master Cm, without ever reactivating the Pmap context. Of course, if any error happens, the automatic bypass is forgotten and the result data goes back to the Pmap context instead, under the expectation that maybe the context could do something intelligent about the problem. In the expected case in which no error occurs, however, a non-local, intracluster memory access requires only one Pmap context activation and three map bus cycles. From the time a processor issues a non-local memory access until it receives the result data, a total delay of about seven microseconds elapses.

4.2.2. More Complex Kmap Operations

Although the memory-access mapping operation is expected to be invoked quite often, the Kmap is by no means limited to providing just that one operation. Generally, the Pmap microprogrammer appropriates some subset of the computer module processor's virtual address space, designating specific addresses which the processor can use to invoke other, less ordinary Kmap operations. With some hardware assistance, the Pmap microcode, by decoding the virtual address of the memory access, determines which particular Kmap operation is desired. A computer such as the PDP-11 which has memory-mapped device control registers invokes input/output operations in exactly the

same manner as on Cm* these special Kmap operations are invoked.

Because each cluster has only one Kmap, the Kmap is the logical processor to perform operations which must be interlocked. If, however, the operation involves a context swap—any operation which performs a Pmap-initiated memory access, for example—some mechanism must be used inside the Kmap to prevent other contexts of the same Kmap from violating the interlock. One technique is to implement an internal semaphore in the data RAM.

Other operations which the Kmap might provide are low-level operating system primitives. In particular, operations which cross domain boundaries, such as a message-transfer operation, and operations which involve much memory traffic, such as a block-transfer operation, are well suited for implementation in the Kmap. The primary objective in microcoding these operations is speed.

4.3. The Interface between Kmap and Kmap

Kmaps communicate with each other via an **intercluster bus**, a packet-switched bus which is jointly controlled by the Linc processors in each of the directly-connected Kmaps. The Linc maintains queues of incoming and outgoing messages, interacts with the Kbus to activate and reactivate Pmap contexts, and provides the local storage for Pmap contexts to construct and inspect intercluster messages. Each Linc interfaces to two independent intercluster busses. Refer to Figure 2-4.

An intercluster message contains up to eight 16-bit words of data, of which all words except the first are totally uninterpreted by the Linc. Each intercluster message is sent from an immediate **source Kmap** to an immediate **destination Kmap**. The number of the destination Kmap appears in a fixed place in the message so that the Linc can determine which messages are sent to its cluster. Intercluster messages are of two types: **forward messages**, which invoke a new context at the destination Kmap, and **return messages**, which return to a waiting context at the destination Kmap. A return message contains the context number of the to-be-reactivated Pmap context in a fixed place where the Linc can find it in order to inform the Kbus. These intercluster messages were designed to be used as a mechanism for implementing remote procedure calls between Kmaps.

When a Pmap context desires to invoke some operation in another Kmap, it prepares a forward intercluster message, instructs the Linc to transmit it on a specified intercluster bus, and then swaps out. The forward message must include the source Kmap number and the originating Pmap context number so that the remote Kmap will be able to send back a return message; there are standard conventions for this information.

When the Linc receives a forward message it has the Kbus activate a new Pmap context to examine the message and respond to the request. Presumably, the message contains some operation code which the Pmap context can identify and act upon. After performing the operation, the context prepares a return intercluster message, instructs the Linc to transmit it on a specified intercluster bus, informs the Kbus that the context is now free, and swaps out.

When the Linc receives a return message it finds the context number indicated in the message and instructs the Kbus to reactivate that context. The context then examines the return message, extracts the result of its requested operation, and continues on with whatever processing it had been doing.

4.3.1. A Simple Multicluster Kmap Operation

From a logical point of view, there is not much difference between the invocation of a Kmap operation by a micro-subroutine call from within the Kmap, the invocation of a Kmap operation by a non-local memory access from a computer module, and the invocation of a Kmap operation by a forward message from another Kmap. Practical considerations abound, of course, but all three are really just different methods of invoking some logical operation. It is often quite convenient that, while in the middle of performing one operation, a Kmap be allowed to invoke a sub-operation which is carried out on a different Kmap. For example, the only physical memory that a Kmap can directly access is that memory which resides in its own cluster; in order to effect an access on memory in some other cluster, the Kmap must send an intercluster message to that cluster's Kmap, asking it to perform the access and to send back the result. The simplest example of such a multicluster Kmap operation is the mapping of a non-local memory access to a location in the physical memory of another cluster. See Figure 4-2.

Some computer module initiates a non-local memory access which activates a context in its cluster's Kmap. This first-activated context is called the **master context**; the Kmap, the **master Kmap**. The master context performs the address translation, discovers that some other Kmap will have to perform the access, sends a forward message to that Kmap, and swaps out to await a reply.

When the message arrives at the **slave Kmap**, its Linc signals the Kbus to activate a new context. This context, called the **slave context**, decodes the request, performs the memory access inside its cluster, and sends the result in a return message back to the master Kmap. The return message identifies the waiting master context, which is reactivated to transfer the result back to the master Cm. As far as the master Cm can tell, the only difference between a non-local memory access which is mapped to a computer module in another cluster and a non-local memory access which is mapped to a computer module in the same cluster is the extra time required for the out-of-cluster operation.

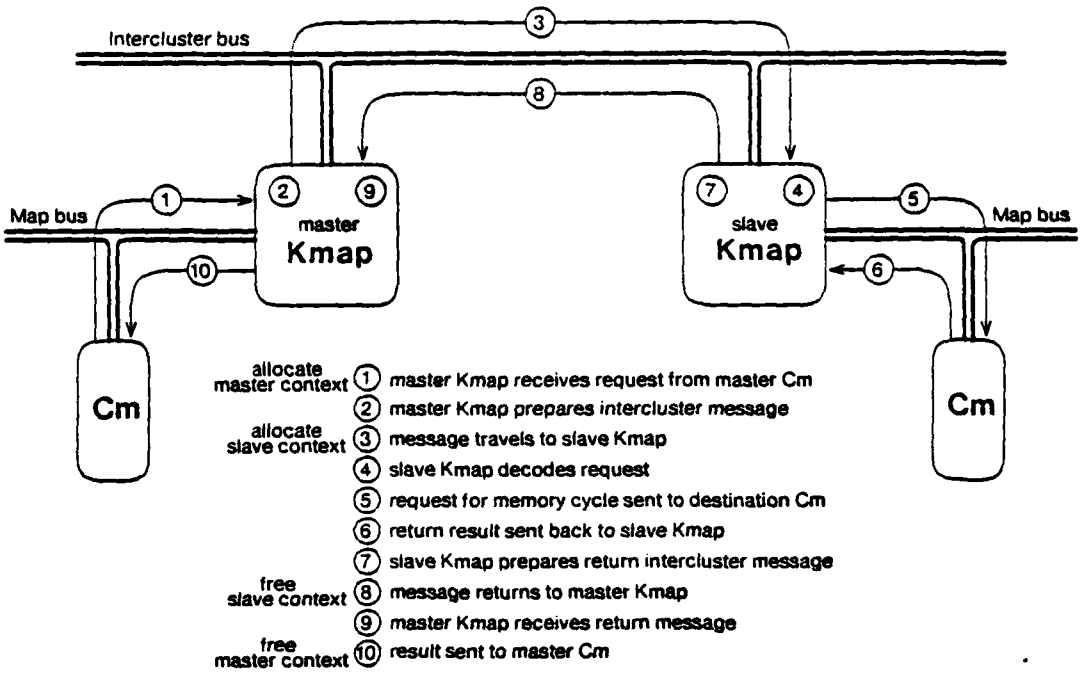


Figure 4-2: The Steps in a Cross-Cluster Memory Access

4.3.2. Forwarding Intercluster Messages

In a configuration of the Cm* system, it is quite possible that two particular Kmaps have no intercluster bus in common and thus cannot send messages directly to one another. Each Kmap connects directly to two intercluster busses, however, and as long as some path through a series of intermediate Kmaps can be found, the two Kmaps in question can still communicate if each of the intermediate Kmaps cooperate by forwarding the message closer to its ultimate destination. See Figure 4-3.

This example differs significantly from the previous example in its style of use of Pmap contexts. In the previous example, contexts are allocated in a nested fashion: the allocation of the master context lasts throughout the entire operation; for a period of time within that allocation, a slave context is also invoked. The potential series of contexts which are allocated at intermediate Kmaps to forward an intercluster message do not wait for a reply, but instead accomplish their entire task by passing a message on to another Kmap.

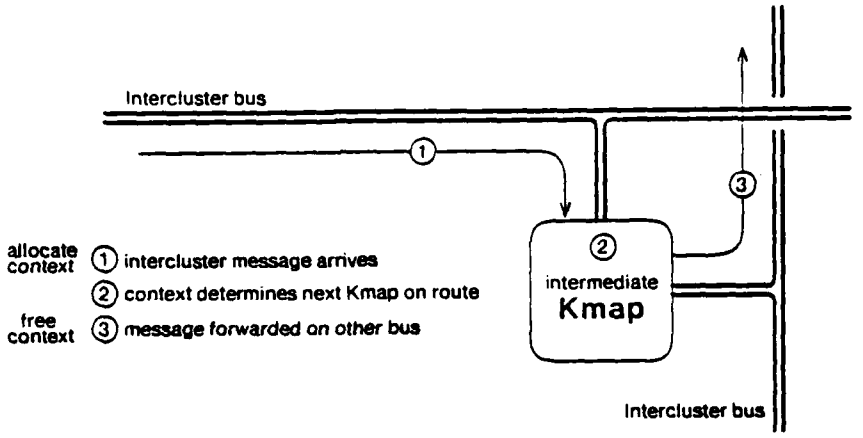


Figure 4-3: Forwarding a Message across Intercluster Busses

4.4. Distributed System Problems

Thus far in this chapter, various difficulties which present some serious problems in designing Cm* microcode have been carefully glossed over. These difficulties arise both from hardware limitations—or, occasionally, malfunctions—and from the very nature of multiprogramming and multiprocessing. Although their particular manifestations relate to the Cm* microprogramming environment, the problems are by no means unique to Cm*, and, in fact, many of these issues exhibit themselves as well in a uniprocessor multiprogramming environment.

4.4.1. Maintaining Consistency

In order to cooperate on any shared task, the individual components of a system must maintain some form of global consistency of state. Even on a multiprogrammed uniprocessor, the design of separate, cooperating processes is a high art. Unfortunately, because of their inherent delay in the propagation of information from one component to another, distributed systems present even more obstacles to the goals of synchronization and maintenance of global consistency.

As an example of this problem in the Cm* system, consider a counter which is updated from time to time by various computer module processors. In the simplest, but incorrect implementation, each processor executes an increment or decrement instruction whenever appropriate. In the Cm* system, the processor must perform such instructions as a memory read followed by a memory write.

Beacuse all memory access requests are carried in message packets on packet-switched busses, there is no a priori guarantee that a request issued by some other processor will not intrude after the read but before the write.

Because this specific problem involves an operation upon memory which resides in only one cluster, it can be solved by requiring that all such operations be performed at the Kmap for the cluster whose memory is involved—a requirement which, since that Kmap would have to perform the memory accesses upon the memory in its cluster anyway, does not really limit performance. This requirement reduces the problem to one of interlocking the several contexts of a single Kmap.

Any operation which involves data which is spread over several clusters, however, is not so easily disposed of. For example, frequently-used data may be cached, for performance reasons, in several clusters. In order to modify such cached information atomically, all possessors of cacned copies must cooperate at least to the extent of realizing that their local copies have been invalidated. Although the microprogrammer can design the microcode to consider some Kmap or even computer module as the authority responsible for maintaining a piece of information, the Cm* hardware provides no implicit central authority for the entire system.

4.4.2. Avoiding Starvation

Within any multiprogramming system looms the danger of **starvation**, the possibility that some process might be so unfortunate as never to manage to acquire some resource it must have, even though that resource is, from time to time, briefly available. Starvation typically results from the simple strategy of repeatedly racing for the resource. Assuming that races are independent, under this strategy the probability that a waiting process will *never* obtain its resource is asymptotically zero. People desire the assurance that the resources their programs need will be acquired within some "reasonable" period of time, however, and the expected waiting time under the simple strategy may be quite unacceptable, especially during periods of severe contention.

In the Cm* system, a single computer module can only process one Pmap-initiated memory access at a time. When the Kbus sees Pmap context α instructing it to request a memory access on a computer module which is currently busy with an access for some other context, the Kbus discards the request and reactivates α with an error indication. Although the simplest thing for α to do would be just to try again, a second attempt would not necessarily guarantee success.

It is possible for two contexts, β and γ, to block every attempt by α to obtain a memory access in a certain computer module, while they themselves are processing a series of different intracluster mapping operations. Every time that α arrives at the front of the run queue, and its request for a memory access is refused because the computer module is busy with a request from either β or γ, the

reactivation of α will be placed at the *back* of the run queue. Given proper, diabolically arranged timing, one of the contexts β and γ can always be on the run queue *ahead* of α whenever the computer module finishes the current request from the other context.

4.4.3. Avoiding Deadlock

Another specter which haunts all multiprogramming systems is **deadlock**, the situation in which a set of processes permanently ceases execution because each process needs a resource which some other process in the set has already reserved. Whereas starvation is usually a probabilistic phenomenon, resulting only in unacceptable delays in execution, deadlock is forever.

The possibility of deadlock arises in any situation in which multiple resources are needed at the same time in order to complete an operation. For example, because of the remote procedure call style of invocation, a cross-cluster mapped memory access requires both a master context in the *master* Kmap and a slave context in the slave Kmap. Now each Kmap only has eight Pmap contexts. If in each of two, previously quiescent clusters eight computer modules suddenly each made an intercluster memory access to the opposite cluster, all eight contexts in both Kmaps could get allocated to the requests from the computer modules, leaving none to respond to the requests which would shortly be arriving across the intercluster bus. No context in these two Kmaps would ever become available because first another context must be allocated to service an intercluster request and send back the result. Deadlock!

4.4.4. Coping with Errors

All software systems face the problem that their supporting subsystems cannot provide absolute, perfect reliability. Occasionally, some operation which ordinarily ought to have succeeded fails in some way. Provided that the supporting subsystem manages to report its failure, it might be possible to attempt some sort of *corrective action*, *either* to repair the current damage or to forestall future failures of the same sort. Although this problem is relevant to all systems, it is even more difficult to address in a distributed system, if it is desired that failure of remote processors or of communication links not kill the entire system, but instead only cause damage commensurate with the failure.

At the Kmap microprogramming level, the supporting subsystem is the Cm* hardware. All of the failures which the Kmap has any *hope of noticing* can be classified as communication errors: transmission errors, which are detected by parity violations on the various communication busses, and selection errors, attempts to address a non-existent memory location, computer module, or cluster, which are detected by timing out after a reasonable pause waiting for a responce.

Generally, these errors are reported to the requesting Pmap context, if one can be identified, as an

error reply to its request. For example, if a map bus error occurs during the reading of the result of a Pmap-initiated memory access back to the requesting Pmap context, the context will receive an error reply instead of the data it was expecting.

4.4.5. Achieving Good Performance

Although possibly content with any level of system performance within a fairly large range, the end user does expect his application programs to proceed toward their conclusion at a reasonable rate. A system which has unbearable performance limitations simply will not be used.

One technique for improving performance on Cm* is to transfer common system operations from the computer module processors into Pmap microcode. The use of this technique is limited by the capacity of the Pmap microinstruction store.

Another technique is to cache in the Pmap data RAM information which is frequently referenced by various Kmap operations but which may be too voluminous to be stored there in its entirety. For example, a complete index of all descriptors for all objects in the system is certainly too large to fit in the data RAM, but since individual descriptors may be referenced frequently and repeatedly, performance can be improved by caching. Caching has its own problems, of course, for when a modification to the cached information is to be promulgated, all of the cached copies, which could be far dispersed in a distributed system, must be rooted out and either invalidated or modified on the spot.

4.5. SMAP: the Simple Microcode *Pradeep Sindhu*

Written during the spring of 1977, SMAP was one of the first microcode systems for the Cm* Kmaps. It provides the bare minimum address-mapping and synchronization facilties needed to allow processors to share memory. No attempt was made in its design to provide protection or generality since its primary use was seen as the base for benchmark programs and diagnostics for the hardware. SMAP has been used extensively for these purposes, as well as for the Algol 68 system and for a number of application programs that have been written for Cm*.

The main function of the simple microcode is to allow each processor to associate any 2048-word physical page in the system with any of the pages in the immediate address space of the processor. A page in the system is identified by its cluster number, Cm within the cluster, and high order six bits of its base address. A page in the immediate address space of the processor is identified by the address space of the processor (user/kernel) and the high order four bits of the processor address, and is referred to as a **window**. Once a window has been associated with a physical page by writing the

mapping tables in the Slocal and Kmap, all references to the window are mapped to the physical page, with the bottom twelve bits of the processor's address serving as the offset into the page. In binding a window to a physical page in its local memory, a processor may choose to have references to the page proceed directly through the Slocal or mapped via the Kmap.

The Slocal mapping table and the mapping table maintained by the Kmap on behalf of a processor are both addressable from the processor. A processor may therefore make any page in the system addressable to itself simply by writing into the appropriate entries in these tables. The microcode does not provide any protection.

A variety of synchronization operations that are useful in making indivisible access to shared data and in coordinating the actions of processors working on a common task constitute the remainder of the functionality of the microcode. There is one operation to indivisibly increment an arbitrary word in memory and four operations to indivisibly decrement an arbitrary word in memory. The decrement operations vary along two dimensions, conditionality and manner of notification. Unconditional decrements are performed regardless of the old contents of the target location; conditional decrements are not performed if the old value of the operation is zero. A processor can select from two methods of notification, synchronous and asynchronous. In the synchronous method the processor examines a result location that is set after the operation completes; in the asynchronous method it is interrupted when the result of the operation is zero.

SMAP contains simple facilities to aid processors in handling errors. When an error occurs during a microcode operation, the Kmap does not attempt to recover from the error. Instead, it collects as much information about the error as it can and stores this information in a place that is accessible to the invoking processor; it then interrupts the processor to signal the error. Recovery from the error is left entirely up to the invoking processor.

There are certain quirks in the implementation of SMAP that prevent it from functioning in a robust way when large amounts of contention are generated at a particular Slocal. The problem has to do with the way in which the microcode handles a busy Slocal condition when it tries to reference the memory of a Cm. The solution used simply waits a fixed amount of time and then retries the reference. Although this is simple to implement, and turns out to be the most efficient solution as long as it works, it runs into problems when large numbers of processors are continually referencing a given Cm. The amount of time a given context waits for a busy Slocal to become free is non-deterministic, and may be large enough to exceed any reasonable time-out period for components that are waiting for the reference to complete. Since the hardware of Cm* provides no way to recognize return requests from operations that have already been timed out, arbitrary damage may result when such a request does return.

John Ousterhout wrote the initial single-cluster version of SMAP; Andy Bechtolsheim added intercluster references, and Pradeep Sindhu made a major revision to incorporate synchronization operations and better error reporting.

4.6. Microcode Measurement Techniques *Ed Gehringer*

The five Kmaps are the central components of Cm*, and the speed of their microcode is critical to the performance of the multiprocessor. Not only does the microcode perform all of the non-local addressing, which can be a major source of contention, but it also implements key functions of the operating system. Since these functions have been included in the microcode to improve their speed, a good deal of attention should be devoted to optimizing the microcode. The aggregate microcode is large—several hundred micro-routines—so it is necessary to focus our efforts at optimization upon the most fruitful areas. This requires us to identify those portions of the microcode which consume substantial execution time.

Toward this end, we have utilized two techniques, each with its own strengths and weaknesses. The first is *tracing* through microcode, counting microcycles and references from the Kmap to the memory of LSI-11's. The second is *real-time measurement*, which involves running a program which performs a given microcode operation repeatedly, usually several million times. The time required for one microcode operation can be calculated by both methods.

The advantages of real-time measurement are these:

- It accounts for queueing delays. Many of the interesting operations in all of the Cm* operating systems are fairly complex, requiring several passes through various queues in the hardware and firmware of the Kmap. Tracing does not take into account any queueing delays within the hardware, and thus trace measurements are only a lower bound.

- It can show the effects of contention. Because tracing cannot model hardware delays, it cannot show how the system slows down as the load on it becomes greater. It is useful to find out, for example, whether a particular operation performs acceptably when all of the Cm's are making references to the same block of data. Only real-time measurement can provide the answer.

Tracing has different advantages:

- It shows which portions of a microcode operation are the most expensive. A real-time measurement yields only one number: the average elapsed time for the operation to be performed. A trace shows how much each microcode subroutine contributed to the total time, and thus indicates where optimization may be fruitful.

- It is easier to perform a large number of traces than a large number of real-time

measurements. A separate program must be written or modified for each measurement which is to be made, whereas with a tracing program (such as the one described below) it is very simple to perform additional traces. Moreover, most microcode operations modify the state of the Kmap or its data RAM in some way, so that consecutive operations in a real-time experiment may follow different paths through the microcode, or even produce errors.

How tracing is performed. Tracing is performed by two programs written by Edward Gehringer in the fall of 1979. The first of these, CYCLES, is used to perform traces of individual micro-subroutines; and the second, RESOLV, uses these traces to calculate how long a Kmap operation takes, by adding together the time to perform a micro-subroutine and all of the subroutines it causes to be called.

When CYCLES is run, it reads in a microcode source file and builds a flow graph. Next it asks the user to select from a menu of micro-subroutines to trace. It then adds up the microcycles and memory references (to the memory of the Cm's) which would be performed when the selected micro-subroutine was executed. At each branch point, it asks the user which branch to take.

It is often useful to perform several traces of the same routine, to account for different values of parameters, or different global conditions. For example, the routine that performs mapped memory references will follow different traces, depending on whether the memory word is local to the cluster or in a foreign cluster. CYCLES thus allows the user to give names to the traces in order to distinguish them. Each time that a micro-routine calls another micro-routine, the user is prompted for the name of the trace of the called routine.

How real-time measurements are performed. A program which invokes same Kmap operation repeatedly is run on one or more Cm's. It is loaded, and then controlled, using the NEST environment, which will be described more fully in Section 5.3.2. This small executive allows the user to specify the number of iterations of the operation, to inquire about the progress of the experiment, and to compute the time consumed by an individual operation from data it displays after all of the processors have finished.

Validation of trace data. Three factors contribute to the time consumed by a Kmap operation: Kmap microcycles, memory references from the Kmap to memory of one of the Cm's, and waiting time due to contention for resources. As noted above, tracing cannot measure contention, but it can give a load-independent lower bound on the time needed to perform an operation. In any event, we expect Kmap and memory contention to be of minor importance for most programs.

To determine how long a Kmap operation takes in the absence of contention, we must know the time consumed by a microcyle and a memory reference. A microcycle takes a constant 157 ns, as it

```
.ru cycles
Input file: msgptr.mic[x336sv30]
Listing file:
No listing file.
Trace file: msgptr.trc
1 2 3 4 5

Here are the routines defined in this file:

  1. FixPtrsSend (26)
  2. FixPtrsReceive (36)

Type the number of the routine to trace, or "quit" to quit. 1

Tracing "FixPtrsSend".  Title of trace: fast
!----> AREG = max pointer value for this segment ;                    0

DEQUESTATE called.  Which trace?
 0.   normal ! We've got the status, etc. ;-                         1
 1.   error ! Error while reading the pointers ;-
Select one of these: 0
FULLMODE:
  [14] ! Buffer mode, buffer not full ;-                             2
  [15] ! Register mode ;-
  [16] ! Buffer mode, and buffer is full ;-
  [17] ! Register mode ;-
Select one of these: 14
!----> FBLATCH = max pointer value ;                                  3

INCPTR called.  Which trace?
DONEINC:
  [2] ! We always come here ;-                                       4

WRITEPTR called.  Which trace? fast
DONESEND:
 0.   normal ! Done writing pointer ;-                               5
 1.  error ! An error occured ;-
 2.  abnormal ! Buffer overflowed ;-
Select one of these: 0
-> HEREQUICKRETURN called.  Which trace?
Exit node.  Totals:
6 microcycles.  Routines Called: DEQUESTATE INCPTR WRITEPTR[fast] -> HEREQUICKRETURN


Here are the routines defined in this file:

  1. FixPtrsSend (26)
  2. FixPtrsReceive (36)

Type the number of the routine to trace, or "quit" to quit.  quit

End of SAIL execution
```

In this example, the user traces the ***FixPtrsSend*** routine, which is used by StarOS to write mailbox pointers when a message is inserted in a mailbox during the ***Send*** operation. All input typed by the user is underlined in this figure. The user entitles this trace **fast**. When the routine being traced encounters a procedure call, Cycles asks the user which trace of the called routine would be followed. For three of the four routines called by ***FixPtrsSend***, the user desires the ordinary unnamed trace to be followed, and thus merely types a carriage return in answer to the question "Which trace?". The microcode performs a **goto** rather than a call of ***HereQuickReturn***, and Cycles indicates this by prefixing the routine name with an arrow (->). Along the right margin, Cycles keeps a running total of the number of microcycles consumed. At the end of the trace, it reports that ***FixPtrsSend*** takes six microcycles to perform, exclusive of any time consumed by the routines which it calls.

Figure 4-4: An Example of Using Cycles

depends on the quartz clock within the Pmap. The trace data tells us how many microcycles and memory references are encountered in a particular Kmap operation; if we know how long the Kmap operation takes, we can thus compute how long a memory reference takes. One calculates how long a Kmap operation takes in the absence of contention by performing a real-time measurement with only one Cm in the cluster running, since there is then nothing to compete with that Cm for the Kmap or for memory.

From our initial real-time experiments, we concluded that a memory reference from a Kmap to a Cm took 4.3 μsec. We then were able to use this value to predict how long other real-time experiments should take to run, based on the trace data. Five dissimilar Kmap operations were tested in this fashion; in all cases the elapsed time was within 2.1% of that predicted, based on our value for a memory reference. These experiments allow us to list values for other Kmap operations without carrying out real-time measurements of those operations, many of which are not amenable to such measurement since they modify the state of the Kmap or data RAM. Microcode-measurement results have helped the operating system projects choose which operations to optimize. The results are reported in several places in the operating system chapters.

5. Standalone Benchmarks

Jarek Deminet

This chapter reports on *standalone* benchmarks. "Standalone" means running without the full support of an operating system. Besides the LSI-11 code of the application programs, the only programs present in Cm* are the Kmap microprograms and, perhaps, a small amount of executive software to facilitate communication with the user. The application software must provide the rest of the functionality normally furnished by an operating system, which means that the programmer is, in effect, programming at a lower level. Further, these programs can be classified as *benchmarks*, since their primary goal is to help evaluate the hardware and firmware. In contrast, the experiments described in the next chapter focus on the applications themselves.

5.1. Goals

Since the previous Cm* Review [Fuller *et al.* 77] and the experiments described there, Cm* has evolved in several ways. This created interest in repeating some of the old experiments and developing new ones in the new, 50-Cm configuration and with different microcode.

The experiments that we are about to recount are less oriented toward hardware performance than the previous ones. The reason is that sufficient data has already been gathered on Kbus bandwidth, memory-reference delay and similar phenomena. The still-open questions are these: what is the optimal number of processors to solve a particular problem, and what are the implications of the Cm* architecture for the algorithms used? The previous results obtained with a 10-processor configuration exhibited nearly linear speedup as the number of processors was increased to 10 for several of the algorithms tested. Will we observe further speedup as the number of processors is increased to 50? The results will also serve as a reference point to measure operating system overhead in future experiments to be performed when StarOS and Medusa are fully operational. As a side effect, some data concerning the reliability of the system can also be gathered.

5.2. Conditions

All the experiments mentioned in this chapter were conducted by Jarek Deminet in the period from May 1979 to March 1980. During that time, both machine configuration and microcode versions changed. The number of available Cm's fluctuated between 40 and 50 and the number of Cm's per cluster varied with time and cluster. The microcode changes were partially due to fixing some bugs

and partially due to improvements and extensions. In particular, a new and much improved version of the StarOS microcode was used in the partial differential equation (PDE) experiment with distributed data.

Even under the same hardware configuration, not all Cm's were always available, since several people were running and testing unrelated programs at the same time. Thus, sometimes different clusters (with different numbers of local Cm's) had to be used for different series of experiments.

All of this makes it difficult to compare results from different runs. An attempt was made to conduct each experiment in as short a time as possible, so that configuration changes would be minimized. In general, each curve presented in this chapter consists of results recorded with the same configuration and the same microcode.

It should be pointed out that the results of experiments have *not* yet been statistically validated. An effort was made to perform a larger number of experiments rather than to repeat each experiment several times. As a consequence, some opinions expressed here about the relevance and impact of individual factors are intuitive rather than statistically validated hypotheses.

In all cases, the experiments were conducted with no other programs running in the same cluster. Thus the only load on the Kmap was that imposed by experiments themselves. Since the number of objects existing at any one time was small, the Kmap was able to keep most of the supporting information in its cache (see Section 4.1).

5.3. Environments Used

The experiments mentioned in the first Cm* Review were conducted with no system software other than I/O routines. Programs were loaded into all Cm's through the serial lines connected to the Cm* Host. As Cm* grew, the number of serial lines was not increased. Cm*/50 still has serial lines connected only to ten Cm's, and only these Cm's can be loaded directly through the Host. Thus, a more sophisticated environment had to be created. In fact, two such environments have been used and will be described in the rest of the section.

5.3.1. Standalone

The first experiments with the PDE and QSORT applications (see Section 5.4) on Cm*/50 used a very simple environment. We defined a set of routines for creating and maintaining a list of available Cm's. All program code was loaded into a single Cm, called the **master**, through its serial line. After initialization, the master transferred parts of the code to other Cm's, called **slaves**, and initialized all

global data, which resided in its own memory. The code to perform these additional functions had to be included in the program for the master. After starting the slaves, the master joined the slaves in solving the problem. At the same time, however, it had to handle clock interrupts and monitor the system in order to report final results. The version of an algorithm using this environment will be referred to as a **standalone** version.

This approach proved to be inadequate for running longer experiments, and inconvenient for smaller ones. The most serious reasons were:

- After the experiment had started, the user was, in effect, cut off from the system. The master, like the slaves, was busy running the application code and could not perform any I/O to the terminal. No interrupts or other events in the processors other than the master were reported.

- The additional functions performed by the master had some influence on the results of its work. In particular, clock-interrupt handling and incrementing the time counter, which was necessary to keep track of the time of the experiment, slowed the processor down.

- To provide all those additional functions, the program for the master had to be different than the one for the slaves, even when the essential functions were same.

- Since the global data was always assumed to reside in the master Cm, it needed to be directly addressable at all times. Cm*'s small address space caused problems. The master memory contained both (master and slave) versions of the algorithm, I/O package, bootstrap and configuration routines and debugger—a total of up to 7 pages. The problem was exacerbated when using the DA Link bootstrap code, which occupies an additional 3 pages. Only 5 pages remained for all the data. This imposed restrictions on the size of problems that could be solved. For example, it constrained the size of the array to be sorted, or the grid for partial differential equations.

5.3.2. Nest

To avoid those problems, an environment called NEST (Nuclear Environment for Software Tests) was created. Its main innovation was to dedicate one Cm, called the **interface module**, for communicating with a user and monitoring other Cm's. This module does *not* take part in solving of the application problem. After initializing global data and starting an experiment, it remains ready for communication with the user. Thus, at any point the user may interrogate the status of the experiment.

The other Cm's, called **remote modules**, constitute a task force for solving a given problem. All remote modules run identical NEST software. Each contains a **supervisor**, which handles interrupts and communicates with the interface module. The supervisor also implements a simple mechanism for multiplexing user processes. The user processes perform the essential work on the problem. The structure of the whole NEST is depicted in Figure 5-1.

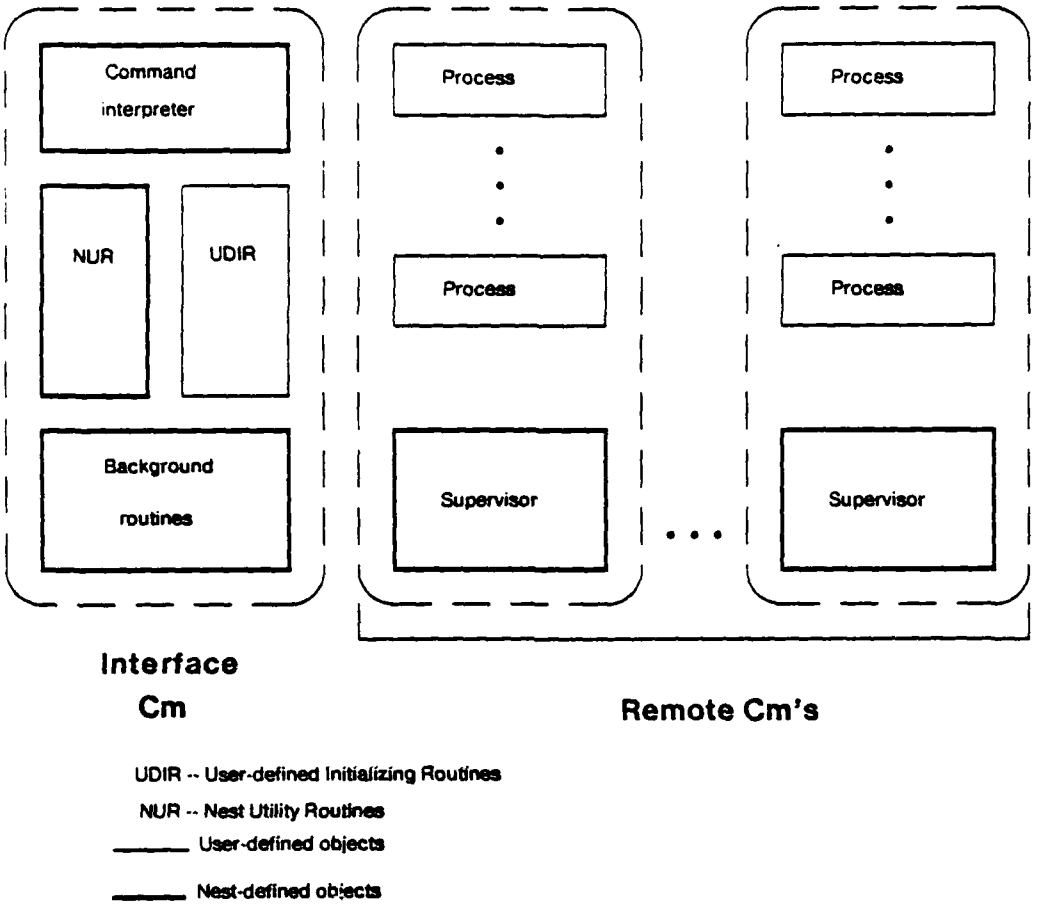


Figure 5-1: The NEST Structure

Because all the functions associated with problem-independent system management are dealt with in NEST, the task of creating a new application program has been made much easier. Moreover, in the interface module as well as in remote modules there is available a set of **background routines**, which define some medium-level memory management operations. This allows a user to write a program with no knowledge about the different low-level memory management operations offered by Cm* itself and the different Kmap microcodes. At present there are three different sets of routines, one for SMAP (Section 4.5), one for the STAROS microcode (Section 7.2) and one for the MEDUSA microcode (Section 8). Both STAROS and MEDUSA microcodes require special data structures in the memory of Cm's. Since NEST runs without the support of the STAROS software, all these structures in the STAROS version are created by NEST itself, according to its needs. Only the necessary objects are used, and the object space is very simple. In fact the only data objects used are full 4K pages. The

MEDUSA version runs with the support of parts of the system software, and all needed structures are created by this software.

Because NEST was to run with different microcodes, not all the tools offered by the STAROS and MEDUSA microcodes could be used, as some of them would have been difficult or impossible to emulate in the others. Only the address-space, synchronizing and initializing operations were used.

5.4. Algorithms Tested

This section describes results obtained with three application algorithms. They will be referred to as QSORT, PDE and NET. Two of them, QSORT and PDE, were modified versions of those written for Cm*/10 by Levy Raskin [Fuller *et al.* 77, Raskin 78]. The versions run with the NEST environment were improved and optimized, compared with those used in Raskin's experiments and in the standalone environment. Hence, results of the three experiments are not strictly comparable. The third application, NET, is new. Each application was implemented as a ***task force***, consisting of several processes cooperating with each other.

The following is a short comparison of features of the applications:

1. QSORT:

   - flexible number of processes
   - one process in each Cm
   - indistinguishable processes
   - functionally identical processes
   - dynamic assignment of tasks to processes
   - global data structures shared by all processes

2. PDE:

   - flexible number of processes
   - one process in each Cm
   - distinguishable processes
   - functionally identical processes
   - static assignment of tasks to processes
   - global data structures shared by all processes

3. NET:

   - fixed number of processes
   - multiple processes multiplexed in a Cm
   - distinguishable processes
   - functionally different processes
   - static assignment of tasks to processes
   - distributed data structures, each shared by a small number of processes.

5.4.1. Qsort

This problem is to sort an array of integers using a modified version of the Quicksort algorithm [Sedgewick 78]. The task force consists of a variable-sized set of indistinguishable processes. Each process runs on its own dedicated processor, so a process may be identified with its processor. The processes share the **sorted array** and another structure, the **stack**. The stack contains descriptors for continuous subsets of the array that have not yet been sorted. As initialized, the single descriptor on the stack describes the whole array. The stack must be lock-protected against simultaneous manipulation by multiple processes.

In each pass a process tries to pop a descriptor for a new subset from the shared stack. If successful, the process partitions the subset into two smaller ones, consisting respectively of all elements less than, and greater than, some estimated "median" value. This "median" is computed as the mean of the first, last and middle elements of the subset. After this partitioning, a descriptor for the shorter of the new subsets is pushed onto the stack, and the longer subset is further partitioned in the same way.

In the Cm*/10 implementation, this action was repeated as long as the longer subset had at least two elements. The algorithm was later improved in the NEST version to avoid too-frequent stack references. If any of the subsets was shorter than some threshold value, it was immediately sorted using the insertion sort instead of being either partitioned or pushed onto the stack. The motivation for this change in a uniprocessor algorithm is given in [Sedgewick 78]. The threshold was set to 10 for most of the experiments (unless explicitly stated otherwise).

In most of the experiments, both the stack and the sorted array were located in the same Cm. There is no good way of distributing them, since different processes have their tasks assigned dynamically so that it cannot be predicted which array element will be accessed by a particular process.

The best speedup theoretically achievable with this algorithm may be computed from the equation

$$\frac{1}{s} = \frac{1}{p} + \frac{2 - \frac{\log_2 p}{p} - \frac{2}{p}}{\log_2 n}, \qquad (1)$$

where

- s is the speedup,
- p is the number of processes, and
- n is the number of elements sorted.

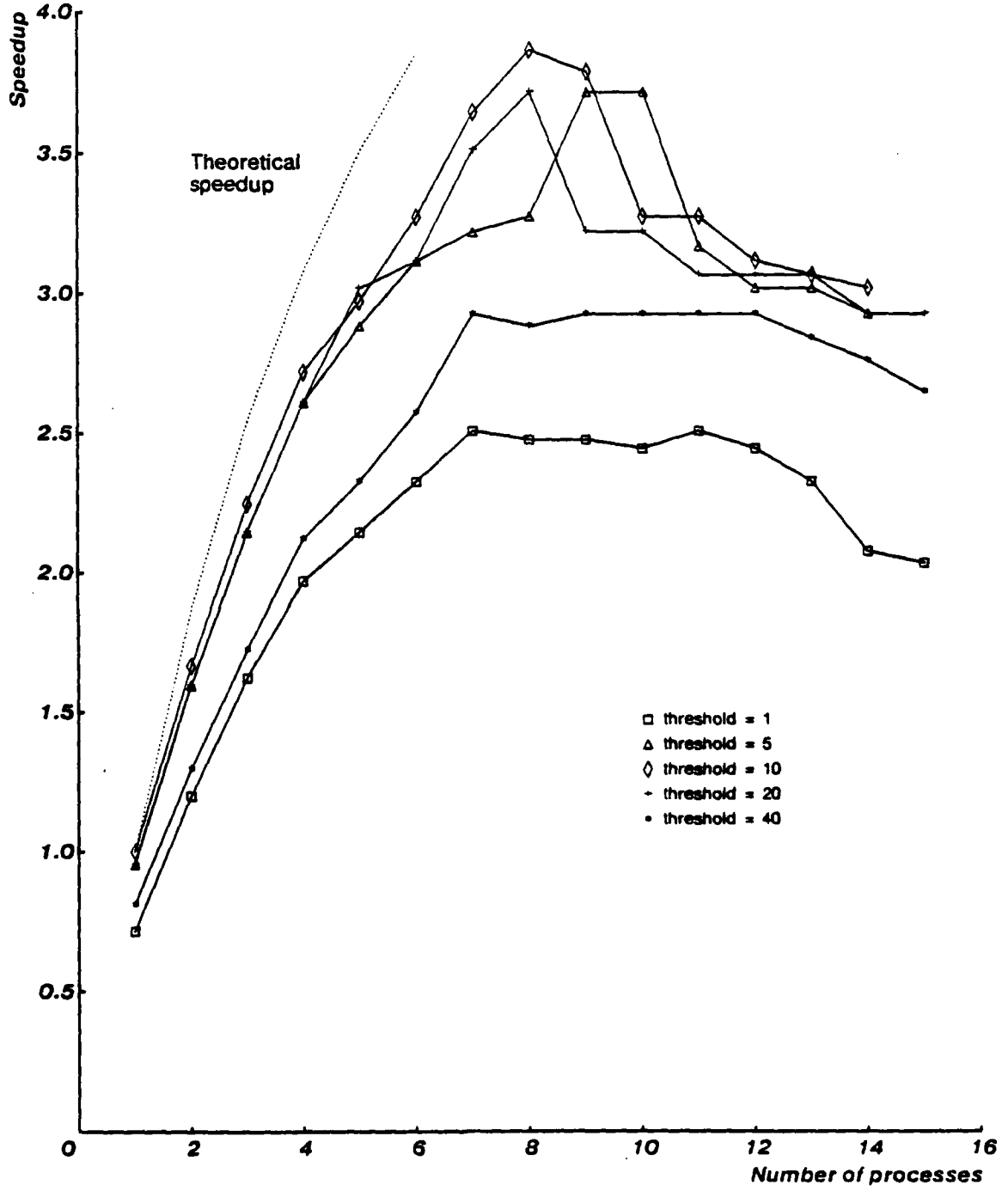


Figure 5-2: QSORT—SMAP, Speedup with Different Threshold Values, 20480 Elements

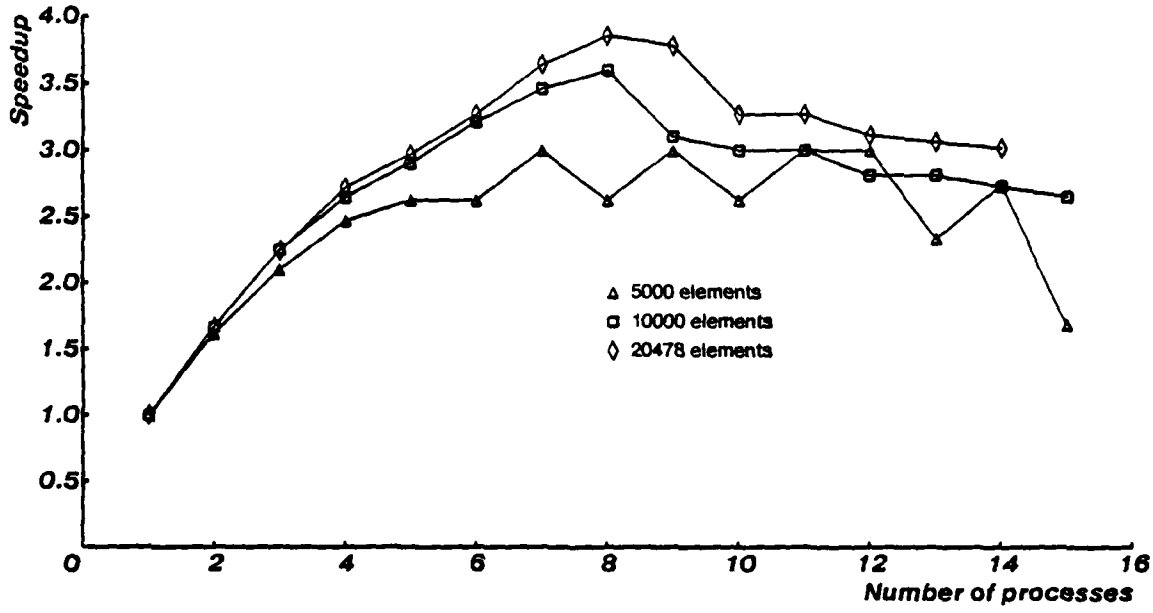


Figure 5-3: QSORT—SMAP, Speedup with Different Data Size, Threshold = 10

For example, for 20480 elements and for 10 processes the expected speedup is equal 5.94. For 25 processes this value is 6.21. For 10000 elements respective values are 4.75 and 5.86. The speedup should continuously grow up to $n/(threshold + 1)$ processes and remain constant for larger numbers. The best speedup obtained with Cm*/10 was about 3.5 for 8 Cm's.

5.4.1.1. Results

Figures 5-2 through 5-5 show the results of the QSORT experiments using SMAP. This microcode was originally designed to serve diagnostic and simple test programs. Its design has some shortcomings and the microcode cannot support heavy use of interprocessor and intercluster communication facilities. For this reason QSORT could not be run with this microcode on more than 16 processors.

Figure 5-2 presents the speedup vs. number of processes for different values of the threshold, using the NEST environment, for 20480 elements. This speedup is relative to the speed of the uniprocess task force with the threshold set to 10. As can be seen, the best results are achieved with the threshold equal to 10. The same result was obtained with a uniprocessor implementation [Sedgewick 78]. The result is, however, still far from the theoretical speedup, especially with more than 8 Cm's. The reason for that is the contention when several processors try to access memory of

the same Cm very frequently. At first, the lock operations were suspected of causing the slowdown. However, if this were true, performance should have been degraded more markedly with the smaller threshold value, as the lock-protected stack activities are more frequent. The figure shows, however, that with the threshold set to 5, the critical point[7] is at 10 processes; whereas for a threshold of 10 it is at 9 processes; and for 20, eight processes. That proves that the observed phenomenon is a result of Kmap contention while servicing the simple memory references, which in the insertion sort are more frequent than in the pure QSORT. In the critical part of the insertion sort, there is one reference to the global array every other instruction, while in the pure QSORT they occur every 3 or 4 instructions.

The results shown in Figure 5-2 were obtained with up to two clusters. The local cluster, which contained the sorted array, had 10 Cm's, so that when fewer than 11 Cm's were used, the configuration behaved as a unicluster configuration. There is no significant performance drop after switching to a two-cluster configuration (except for threshold = 5). This can probably be attributed to the already poor performance, due to the Kmap contention previously noted, of a large unicluster configuration.

Figure 5-3 shows a comparison of results with different data sizes. The speedup is relative to the uniprocess task force for each array size. Better speedup with the larger sorted array is in accordance with theoretical expectations. The results for the smallest array are very irregular since the time of that experiment was very short and very small absolute differences could cause a large relative difference.

Figures 5-4 and 5-5 show a comparison of the best results obtained with the new NEST version (threshold = 10) with the old standalone version (effective threshold = 1). Figure 5-4 presents the best speedup values achieved with each version. Because of address space limitations the maximum data size in the standalone version was less than half of that available with NEST. This is one source of better performance of the NEST version. Figure 5-5 shows the absolute time for both versions with the same data size (10000 elements). As can be seen, even then the NEST version is roughly twice as fast as the standalone one, especially for a small number of processes. This is due to the optimization of the version of the program run with NEST.

Figures 5-6 and 5-7 show results of QSORT with the STAROS microcode. Figure 5-6 presents the speedup vs. number of processes for different threshold values (for 20480 elements). In this experiment the local cluster had 8 Cm's. Kmap saturation occurs around 6 processes, independent of threshold values, with the exception of threshold = 1. There is no significant performance drop when

[7] That is, the point where the performance begins to decrease significantly with growing numbers of processes.

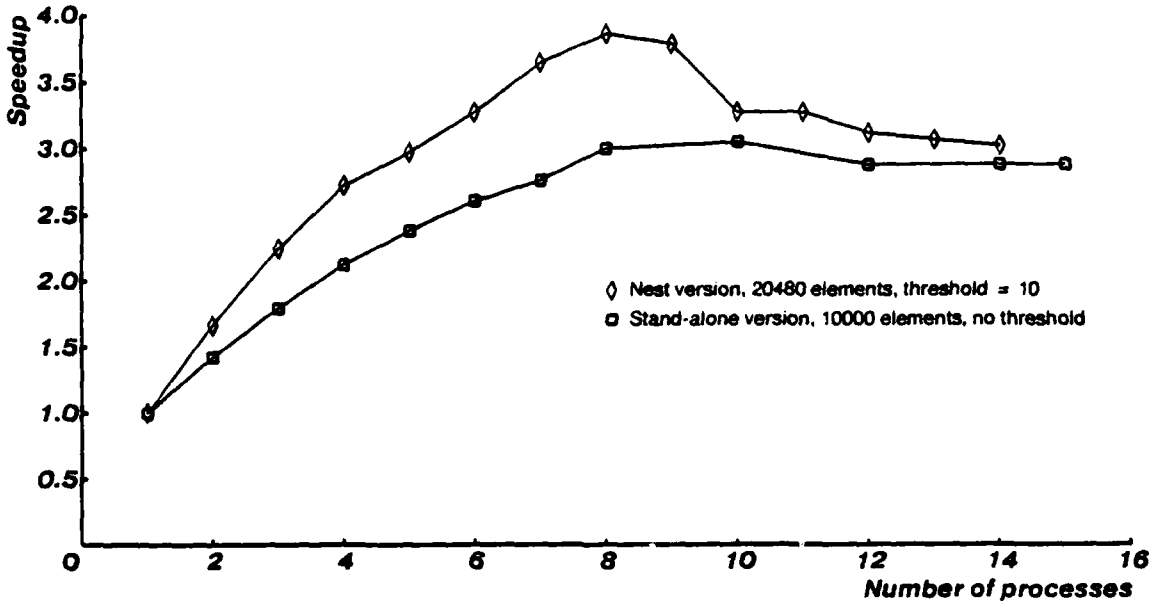


Figure 5-4: QSORT—SMAP, Speedup in NEST and Standalone Versions

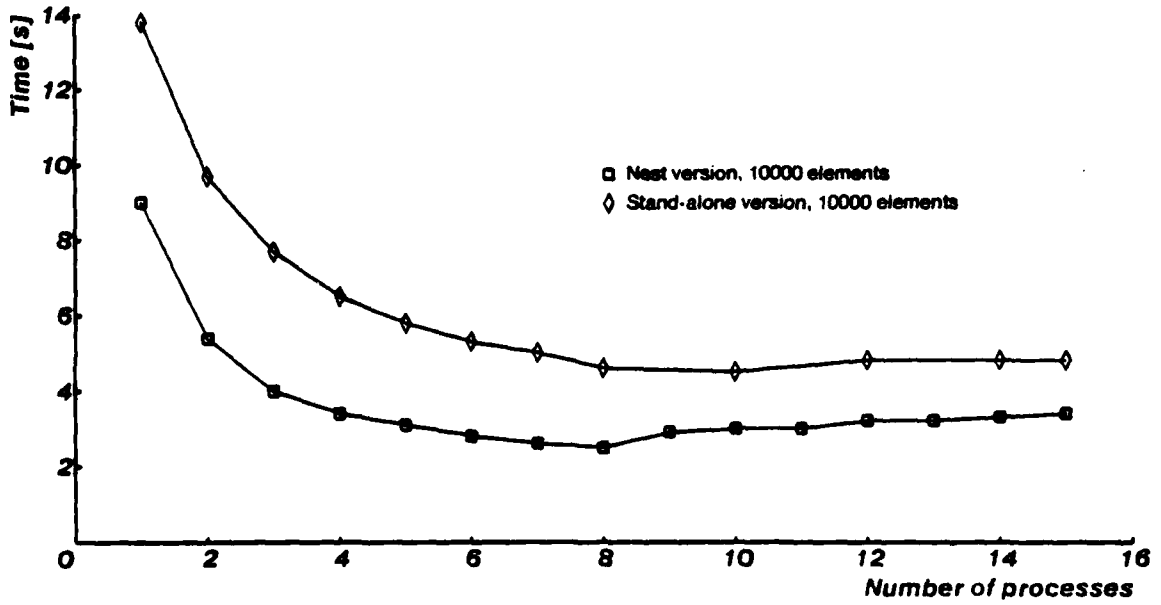


Figure 5-5: QSORT—SMAP, Absolute Time in NEST and Standalone Versions

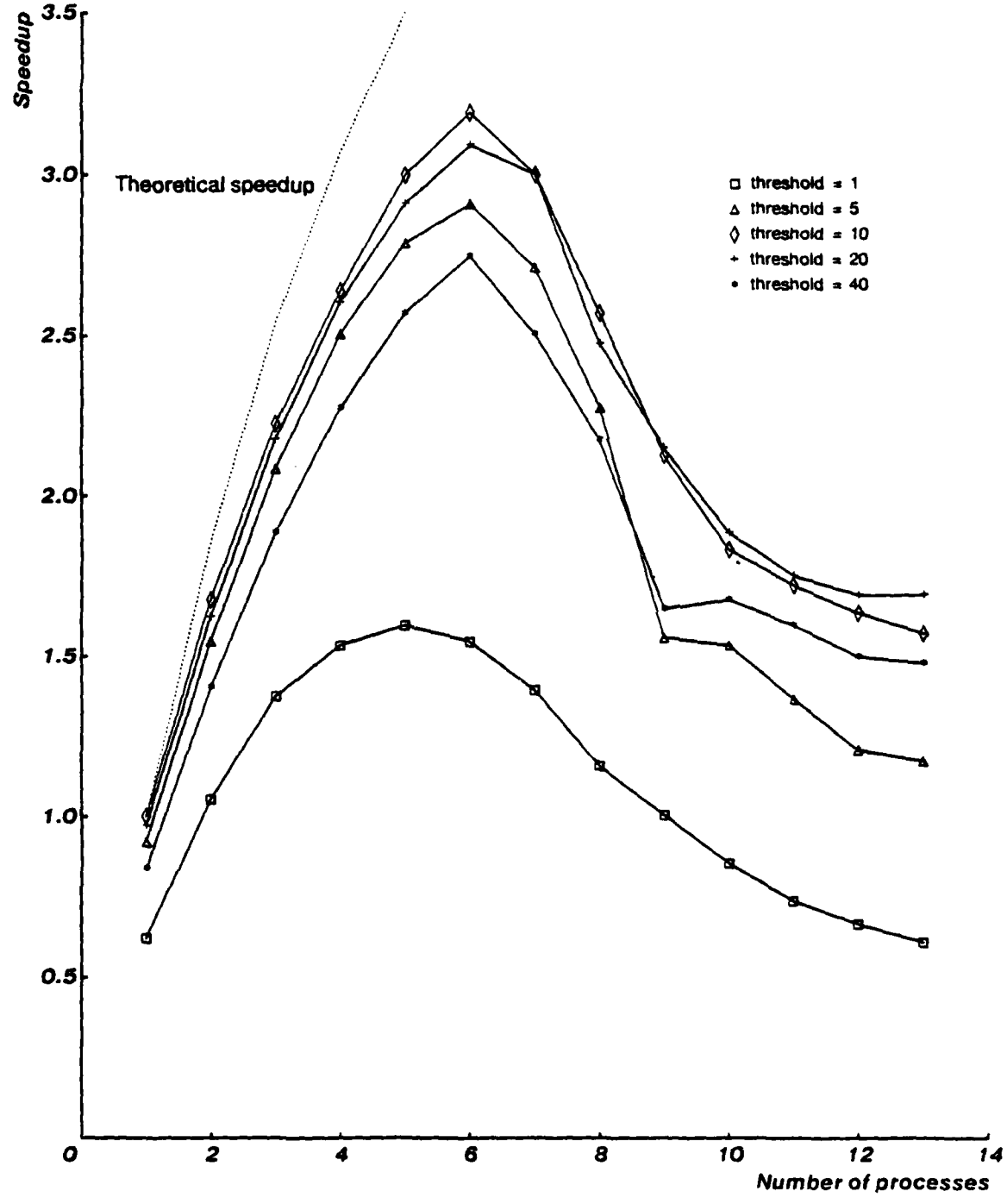


Figure 5-6: QSORT—STAROS, Speedup for Different Threshold Values, 20480 Elements

crossing the intercluster boundary (between 8 and 9 Cm's). This figure may be compared with Figure 5-2. QSORT using the STAROS microcode exhibits slightly worse performance than QSORT using SMAP. The reason is that the functions of the STAROS microcode are more complex and time-consuming than those of SMAP (see Chapter 10).

Figure 5-7 corresponds to Figure 5-3 and shows the speedup vs. number of processes for different numbers of elements to be sorted. As expected (see Equation 1, page 64), the smaller the size, the worse the speedup achieved. The shape of all graphs, however, is essentially the same. Again, for a large number of processes, performance with larger thresholds becomes worse than with smaller ones (note crossing graphs for thresholds 20 and 10, and 5 and 40). This demonstrates that the degradation is caused by Kmap contention during simple memory references. The larger the threshold value, the more frequent these references are.

Figure 5-8 presents the results obtained with the MEDUSA version. The best achievable speedup is between 3.0 and 3.5, similar to the STAROS version and 20% less than with SMAP. The optimal threshold value is 10, just as in both other versions. It should be noted that the best speedup for threshold = 5 was achieved with a smaller number of processes than the one for threshold = 20. That means that the speedup is bounded by the cost of simple memory references rather than by the cost of synchronization, as was discussed above.

Several experiments with both STAROS and MEDUSA microcodes were also performed for a large number of processes (up to 40). The results showed constantly dropping performance for all threshold values and data sizes.

5.4.1.2. Conclusions

The best speedup achieved with QSORT was about 4 for SMAP and 3.3 for the STAROS and MEDUSA microcodes, for less than 10 processes in a task force. Those results suggest that the QSORT algorithm is not suitable for distributed processing with a large number of processes, at least using Cm*. In theory, the best result should be achieved for the number of processes equal to $n/(threshold + 1)$ where n is the number of elements sorted. In practice the critical value is much smaller. The reasons for this are probably associated with the structure of the computer. In this algorithm a remote reference to the global data occurs, in some critical parts, every 3 or 4 instructions. Since all shared data is located in one Cm, all those references are directed to this Cm, causing heavy contention. Thus, the effective reference time is much longer than the theoretical inter-Cm reference time measured with no saturation or contention.

Several experiments have also been performed with the global data distributed between different Cm's. This does *not* improve locality of the references, since tasks are assigned to processes in a

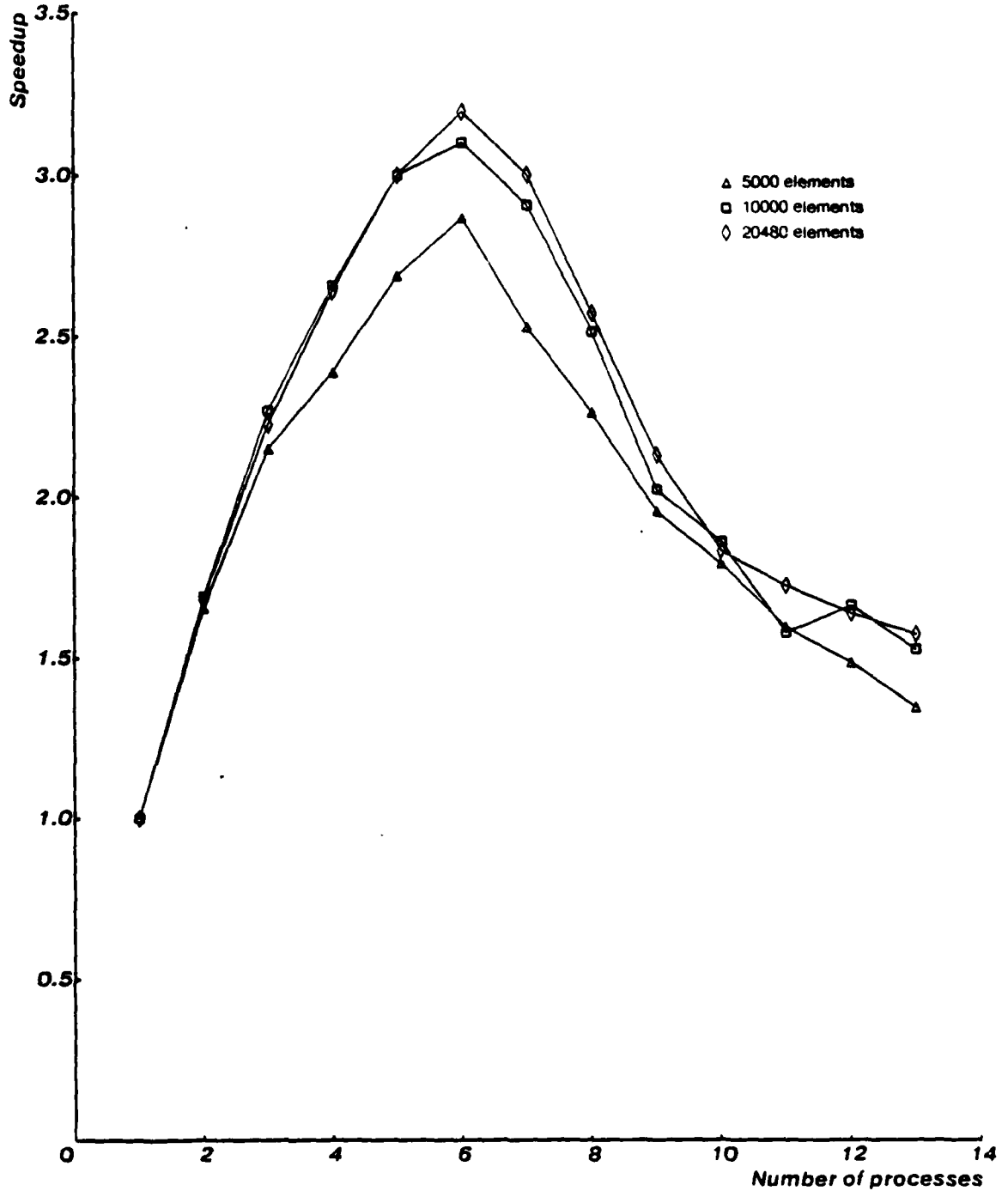


Figure 5-7: QSORT—STAROS. Speedup for Different Data Size, Threshold = 10

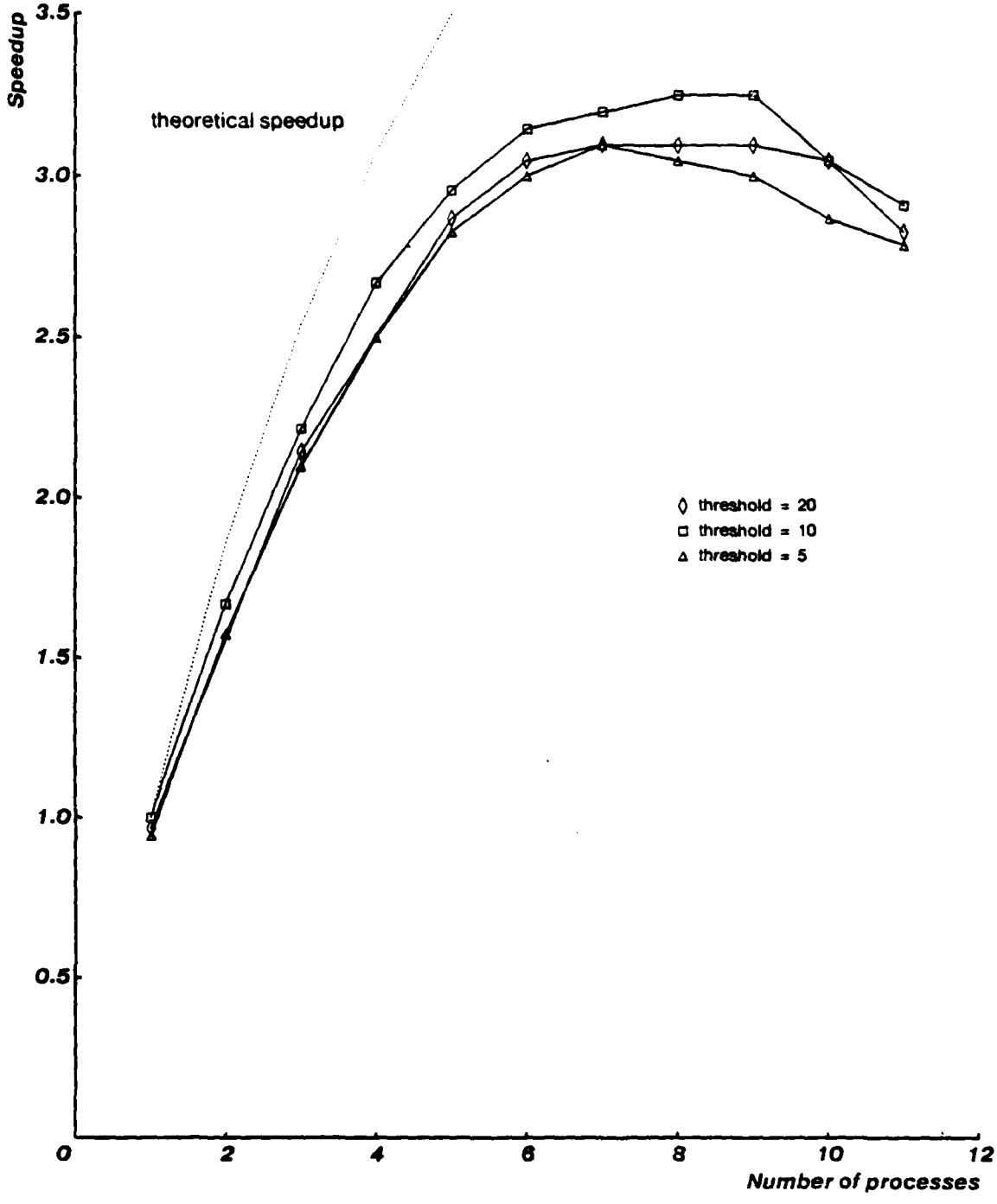


Figure 5-8: QSORT—MEDUSA, Speedup for Different Threshold Values, 20480 Elements

random way and most of the processes still access non-local data. Distribution makes no significant difference.

5.4.2. PDE—Partial Differential Equations

In this application, the objective is to solve Laplace's partial differential equation with given boundary conditions (Dirichlet's problem) by the method of **finite differences**.

The equation

$$\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} = 0 \qquad (2)$$

is solved for points of an *m*-by-*n* rectangular grid. The solution is found iteratively. On each iteration the new value of every element is set to the arithmetical average of the values of its four adjacent neighbors.

Each process of the task force solving this problem runs on its own dedicated processor, so a process may be identified with a processor. Each process performs the iteration for a fixed, continuous subset of the grid array, which will be called a **task**; thus the processes are distinguishable.

There are many possible ways to synchronize the processes and buffer the data. Raskin [Raskin 78] described four of them. The **purely asynchronous** method was selected for these experiments. It is most efficient and allows—theoretically—linear speedup. In this method, each process updates the value of each point using the current values of adjacent points directly from the shared array and places the new value immediately into the array, so that it is available for other processes. This reduces the working space because no buffers are needed and also assures that the newest—and probably the best—approximation is used as soon as it is computed. The only variable that has to be locked against simultaneous use is one that records the number of processes that haven't finished their tasks yet. This variable is accessed only once per each iteration.[8] The best speedup achieved by Raskin with this version was 7.2 for 8 Cm's.[9] We were interested in whether similarly good results could be obtained with a much larger configuration as well.

---

[8] Speedup of a task force in this section will be relative to the uniprocess or task force for the same data size and under the same conditions, unless explicitly stated otherwise.

[9] For comparison, the other, more synchronized versions, had speedup from 5 to 6.3 for the same number of Cm's. Also, the absolute time in this version for a small number of Cm's was up to 50% less than in some of the other versions.

The version of the program used for standalone experiments was similar to that used by Raskin. The NEST version was improved and optimized. Because only one synchronization method was used, the code performing some actions required only by the other modes could be removed and the program could be restructured.

5.4.2.1. Results

The number of experiments with PDE was greater than with the other programs because in this application several aspects of behavior may be measured more easily than with other benchmarks. During a given experiment, the same amount of work is performed by each process during one iteration. This allows us to measure the relative speed of different processes in different configurations. The theoretical speedup is linear despite the size of the grid; therefore experiments with different data sizes may be easily compared.

The largest array available with the standalone version was 20-by-20. The run time of the uniprocess task force was approximately 4 minutes in that case. The NEST version was used mainly with 40-by-40 or 150-by-150 arrays, with a run time of 3 minutes and 13 hours respectively. This shows how the magnitude of the computation to be performed increases when the data size grows significantly. The smaller time for a 40-by-40 array in the NEST version than for a 20-by-20 in the standalone version was caused by the generally improved performance of the former one.

Figures 5-9 through 5-16 present results obtained with SMAP. Figures 5-17 through 5-20 show corresponding results with the STAROS microcode.

Figure 5-9 shows the comparison of the speedup in three different experiments varying the environment and the grid size. The theoretical linear speedup is shown as a reference line. As can be seen, the best speedup was achieved with the standalone version. It is almost linear, with a coefficient of 0.77. This is caused, ironically, by the inefficient implementation of the process algorithm in this version. Since the number of global data references remained the same in the NEST version while the number of other operations decreased, the time between the references decreased as well. This led to heavier contention and greater communication overhead relative to the program runtime. Figure 5-10 shows the comparison of the absolute run time of two experiments with the same data size in both standalone and NEST versions. Note that the vertical scale is logarithmic. At any point the NEST version is up to 4 times faster. These results show how careful one has to be in evaluating different algorithms or programs for parallel processing. The program that offers the best speedup may be otherwise ineffective and the absolute time may be worse than with some other algorithms.

Figure 5-11 also presents speedup for small data sizes (30-by-30 and 20-by-20). It is not obvious

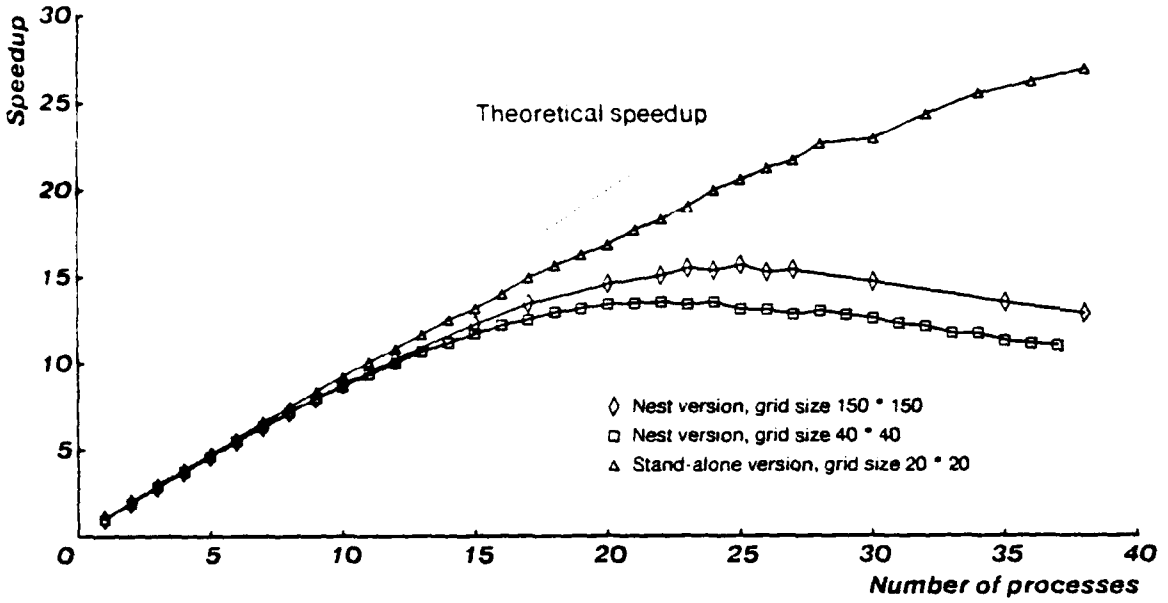


Figure 5-9: PDE—SMAP, General Comparison

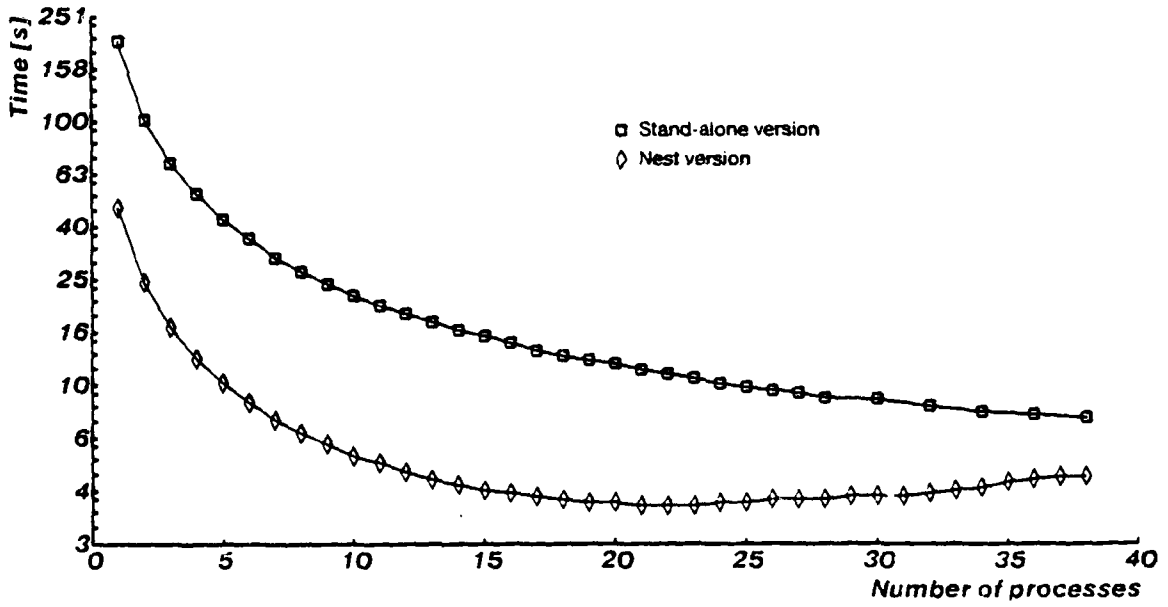


Figure 5-10: PDE—SMAP, Absolute Time in NEST and Standalone Versions

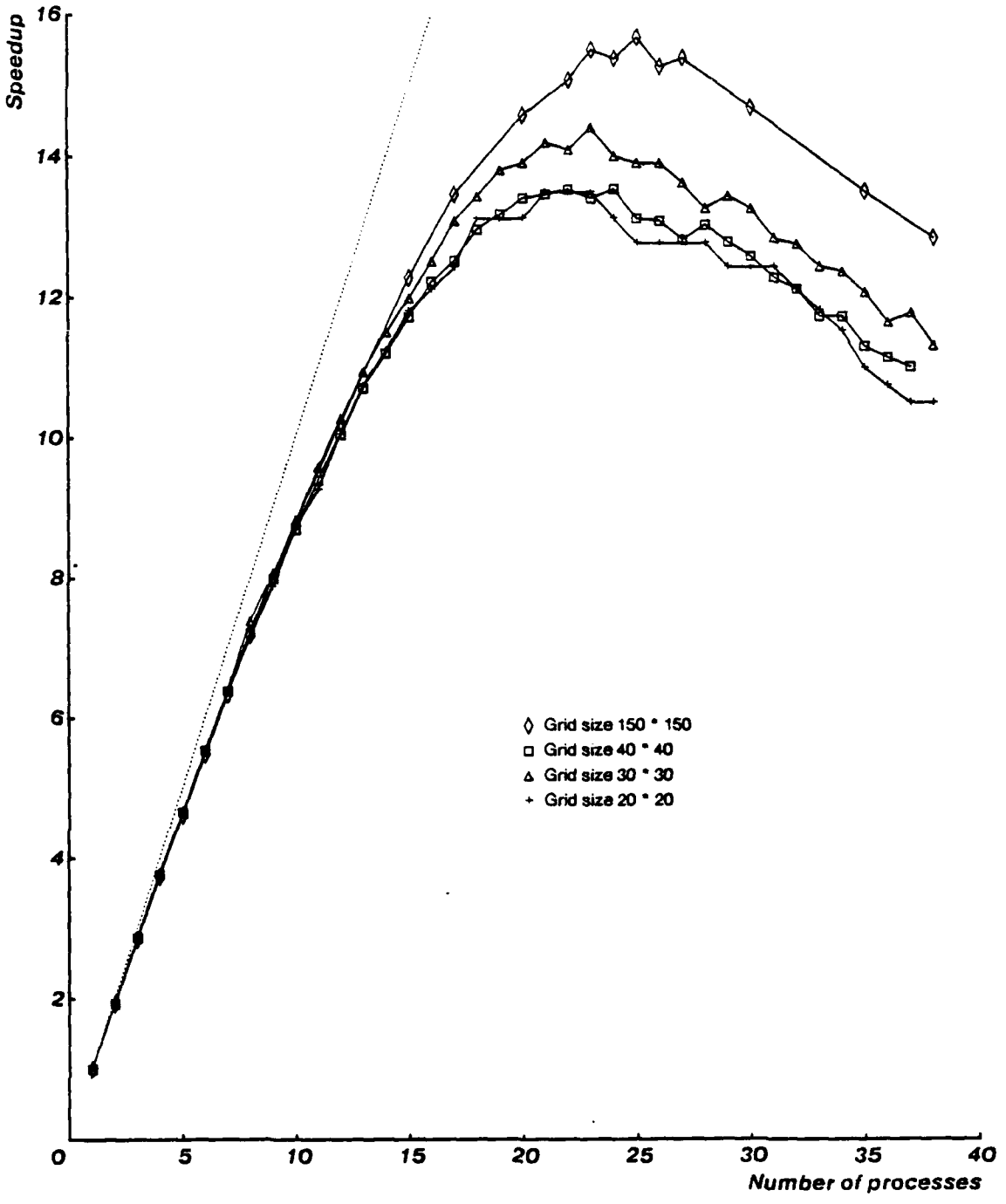


Figure 5-11: PDE—SMAP, Speedup for Different Data Size

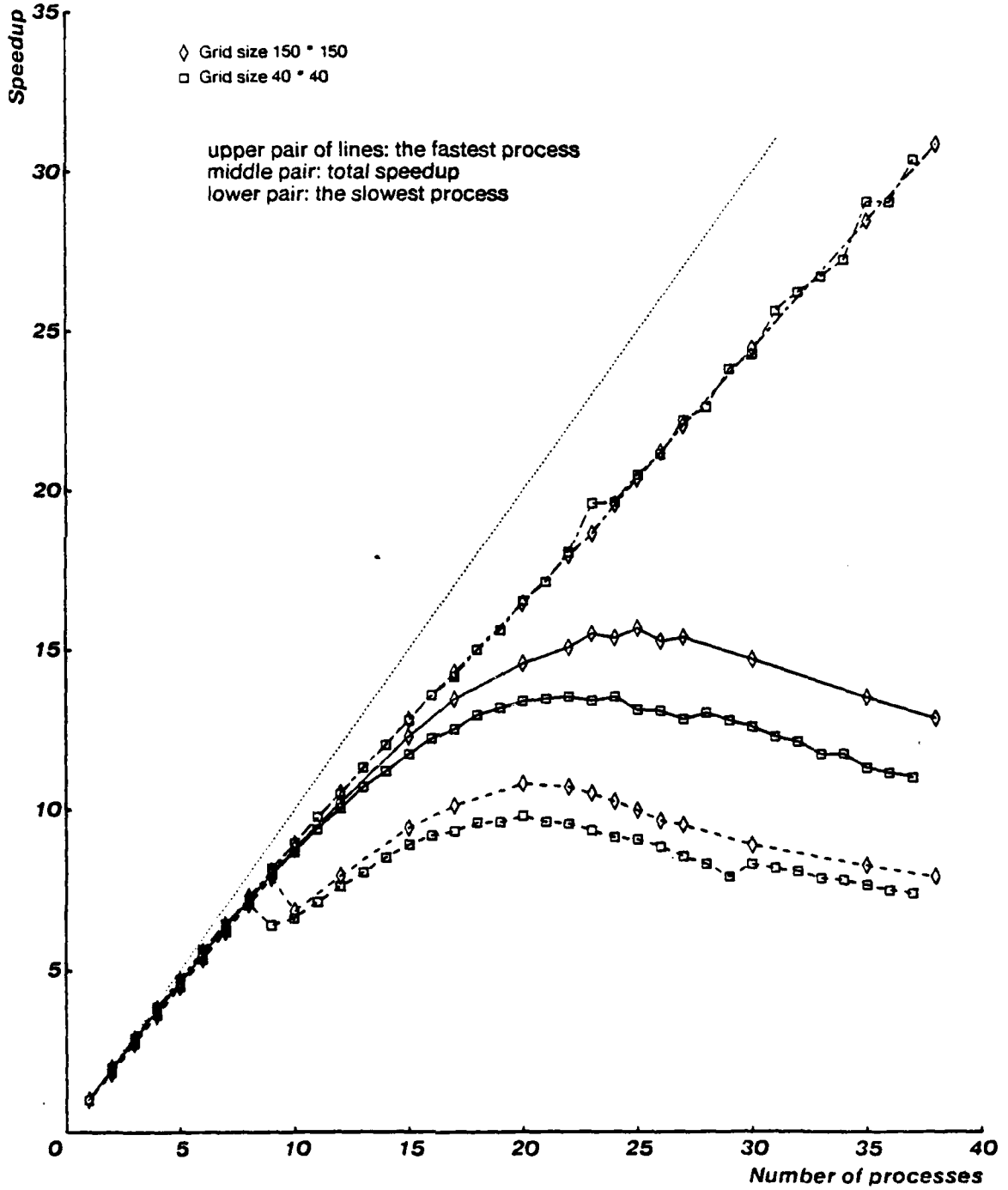


Figure 5-12: PDE—SMAP, Adjusted Speedup

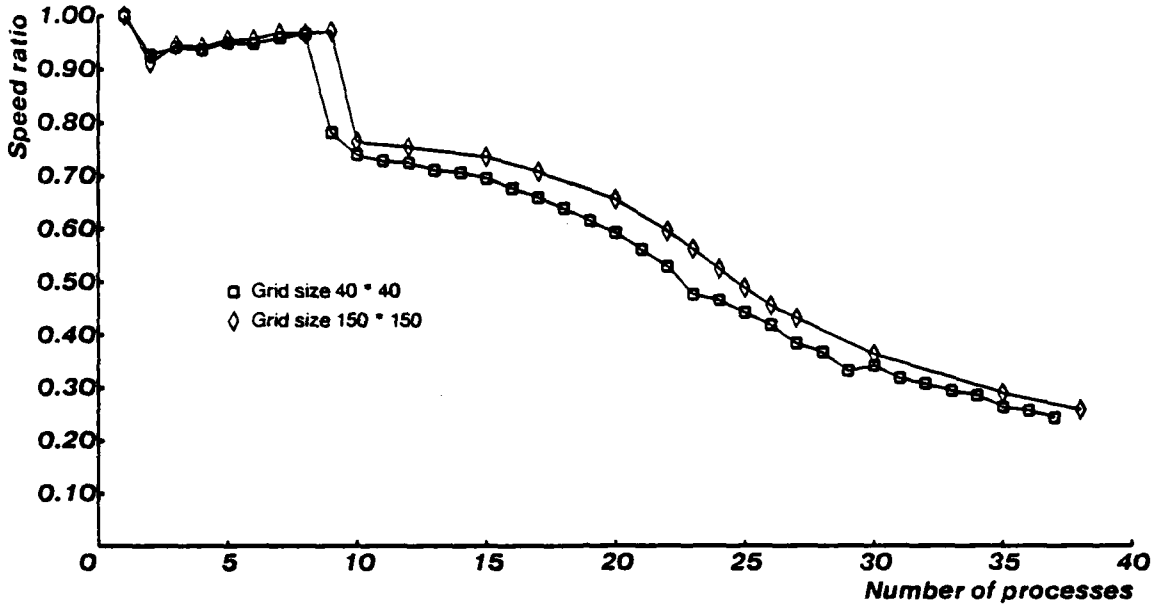


Figure 5-13: PDE—SMAP, Speed Ratio of Different Processes

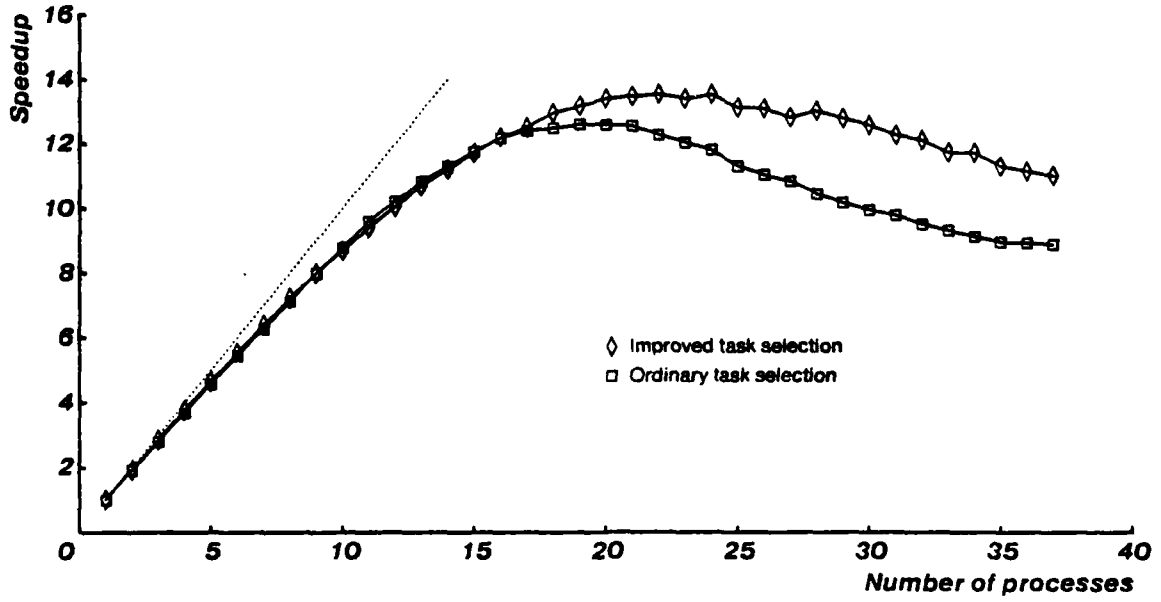


Figure 5-14: PDE—SMAP, Speedup with Different Task Selection

why the speedup depends on the size of data. Intuitively, any irregularities such as different number of references to the boundary-condition data set, which requires slightly more computations than references to the grid points, play a more important role with the smaller data. Because the total time is determined by the time of the slowest process, such irregularities may slow down the whole task force. Besides, updating of the lock-protected variable, performed once after each iteration, is more frequent with the smaller data.

Figure 5-12 presents some results of a more detailed study of this problem. For each data size, it presents three different curves. One of them shows the total speedup of the task force as a whole. The other two show the adjusted speedup of the slowest and the fastest processes in the task force. Each of those values was computed as a ratio of the time of one iteration for the uniprocess task force to the time of one iteration for the slowest (or fastest) process in the multiprocess task force. One may say that this is the speedup of a hypothetical system, each component of which runs at the same speed as the measured process.

As can be seen, there are no irregularities in the speed of the fastest process (which runs in the Cm containing the global data). Its speedup is still not as good as theoretically possible; however it is linear with a coefficient of about 0.87.

On the other hand, the speedup of the slowest process is not so regular. It is almost linear when the configuration is unicluster. After crossing this point there is a significant drop in performance (in one experiment it takes place at 9 processes, in another at 10). From this point on, the speedup increases slowly and finally starts to decrease. Those changes are reflected in the total speedup. As long as the number of the processes running in the local cluster (that is the one containing the global data) is larger than the number of other Cm's, the speedup is close to linear (up to about 20 Cm's). After that, it resembles the speedup of the slowest process.

Figure 5-13 shows the speed ratio of the slowest vs. the fastest process in the configuration. Again, as long as all references are intracluster, no significant performance degradation is observed and all processes run at least at 90% of the speed of the local process. This result is in full accordance with the previous results from the unicluster Cm*. In the multicluster configuration the results are much worse and with 38 processes running, some of them waste up to 70% of their time due to the remote-access overhead. Since this overhead grows with the number of processes running in parallel, significant contention must have been encountered. Most probably, this is the contention in the Kmap when several Cm's want to access the same destination Cm. Since the Slocal of that Cm can serve only one reference at a time, the others must wait. The time for a reference to be processed is determined by the Kmap, Slocal and LSI-11 times. It is very unlikely that the latter two experience much contention, as otherwise they would significantly slow down the local Cm as well.

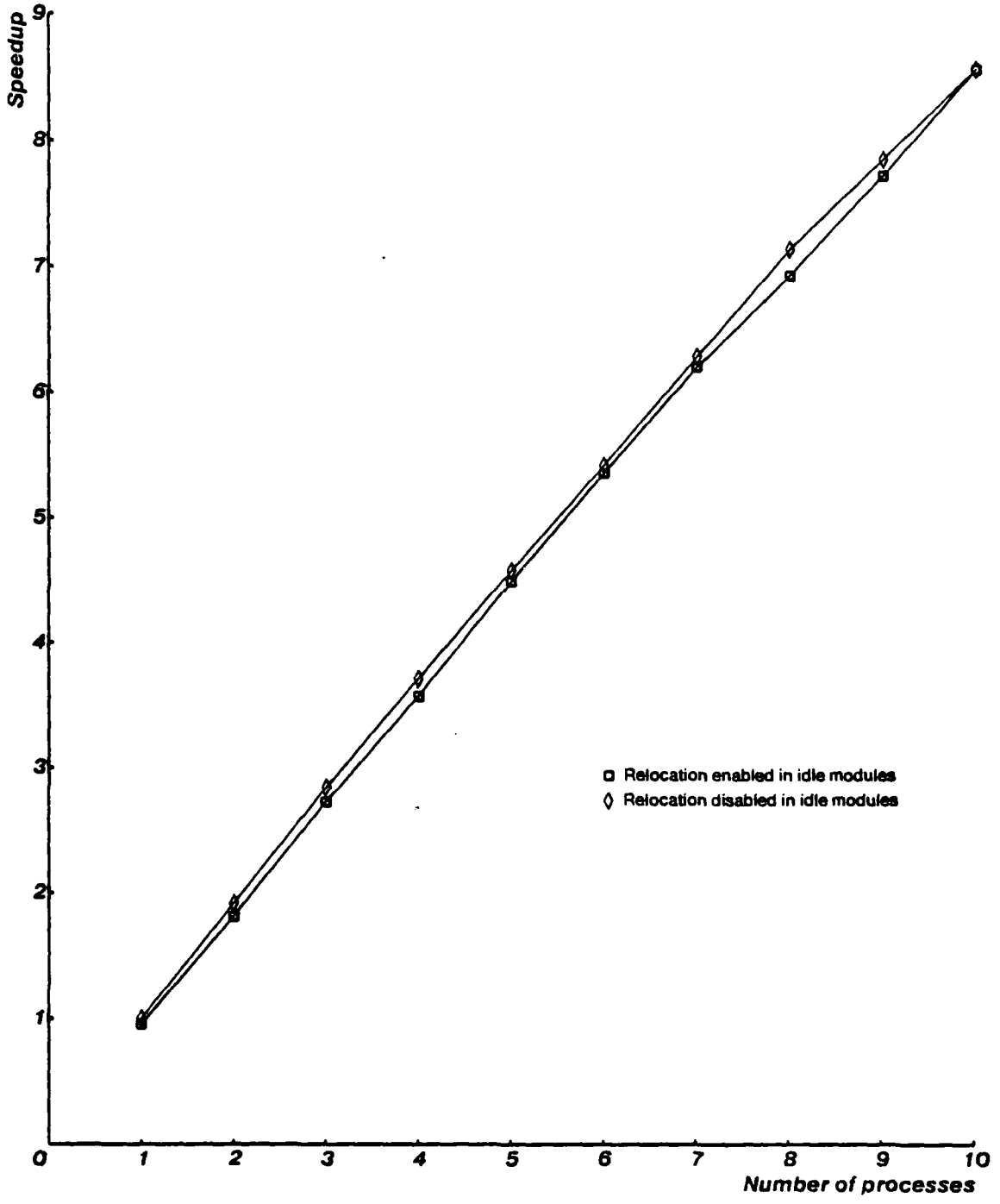


Figure 5-15: PDE—SMAP, Speedup with Different Behavior of Idle Cm's

In the original Cm*/10 version of the algorithm, the assignment of different tasks to processes was either random or based on the number of the Cm in the pool of available Cm's. This was not suitable for a multicluster configuration. In fact, not all tasks need to perform the same amount of work during an experiment. The middle tasks require more iterations than the ones that are close to the boundary, so they should be solved by the fastest processes. In general, the closer the task is to the middle of the array, the closer to the global data (in terms of reference locality) should be the process solving it. A version that satisfies this condition will be referred to as an **improved task selection** version.

Figure 5-14 compares the results obtained with the two methods of task selection. The results with the improved task selection are significantly better when the number of processes is larger than 16 (more than two clusters involved). This corresponds to the point when the speed of different processes becomes significantly different due to rising intercluster reference time (Figure 5-13). At 35 processes, improvement is about 20%.

The LSI-11 microcode provides another way of measuring intracluster memory reference degradation. When an LSI-11 is halted, it runs the ODT microcode communicating directly with a console (the Host serial line in Cm*). The microcode performs a busy wait, trying to read the console register on its LSI-11 bus in a continuous loop. If relocation is enabled, all those reads are serviced by the Kmap. Figure 5-15 shows the results of two different runs. In one of them, relocation in the idle Cm's was disabled; in the second one, it was enabled. Note that in this case the speedup in both versions is relative to a uniprocess task force with relocation disabled. As can be seen, the difference, although visible, is very small. This is one more proof that a Kmap may service intracluster references with no significant performance degradation when no *contention* occurs (each processor makes references to its own Cm).

To alleviate the problem of contention observed in the PDE experiments already described, an attempt was made to distribute the global data. This was possible only if the data was larger than a memory page because a page is the smallest unit of memory that may be effectively allocated in a Cm. Figure 5-16 shows the results for a 150-by-150 array. The pages containing the array were distributed between clusters to maximize intracluster locality of references. This distribution was completely invisible to processes themselves. The task selection was such as to increase locality of references as much as possible. The improvement is significant (compare with Figure 5-12 or Figure 5-9). The speed of the slowest process in the 37-process configuration was about 68% of the speed of the fastest one, as opposed to 28% in the non-distributed version.

The performance drop at about 20 processes was caused by the distributing algorithm, which is far from optimal. The distribution is only between clusters and not between Cm's in a cluster. Because data is distributed uniformly between clusters, when the number of processors running in different

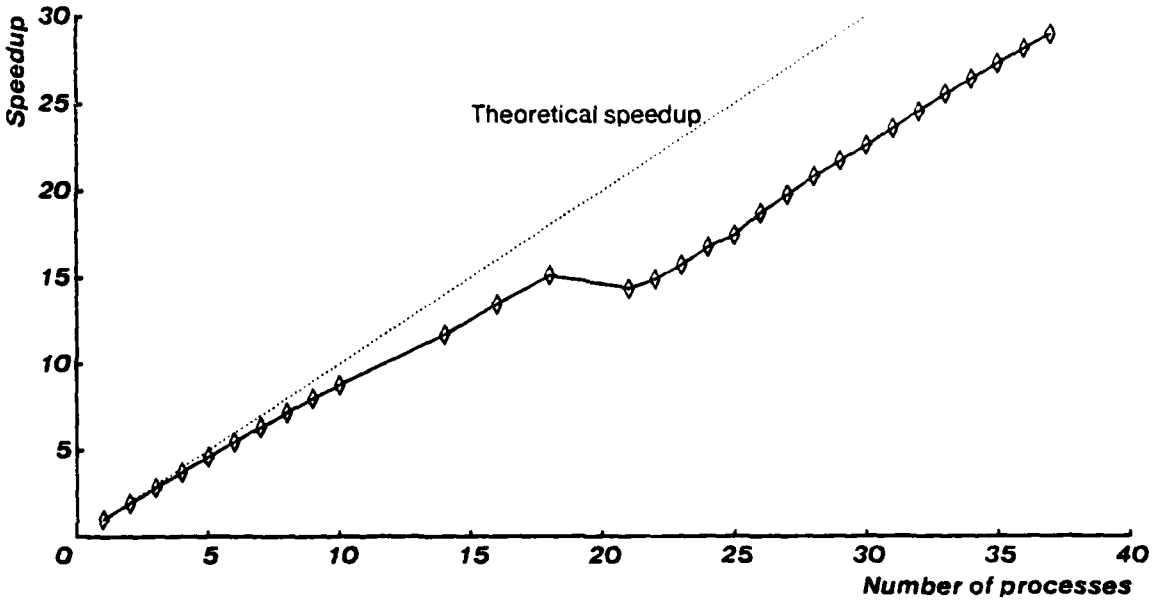


Figure 5-16: PDE—SMAP, Distributed Data

clusters differ, the number of intercluster references may increase. This **crossover phenomenon** is responsible for the performance drop after changing, for instance, from two clusters and 20 processes to three clusters and 21 processes, with the extra cluster hosting only one process.

The next few figures show results obtained with the STAROS microcode using the NEST version. Figure 5-17 shows the speedup in an experiment with the 40-by-40 array with ordinary and improved task selection. The speedup, linear as long as the configuration is unicluster, drops after crossing the cluster boundary (at 10 processes) and continues to drop afterwards. The performance of the improved-task-selection version is slightly better, especially in the decreasing region. This figure presents additional evidence that the main problem with a large configuration is Kmap contention. The STAROS microcode is slower than SMAP: comparison of Figure 5-17 with Figure 5-14 shows that the performance drop is more pronounced in the case of STAROS.

Figure 5-18 shows the speed ratio (slowest/fastest Cm) for the STAROS microcode. The performance drop is very large, reaching 95% in the 36-process configuration (that is, the slowest process runs at 5% of its nominal speed). A very rough estimation[10] gives a value of 1200 μs per one

[10] $(t_r - t_l)/n$, where t_l and t_r are the run time of one iteration in the Cm containing the shared data and the Cm in a different cluster, respectively; and n is the number of memory references per one iteration.

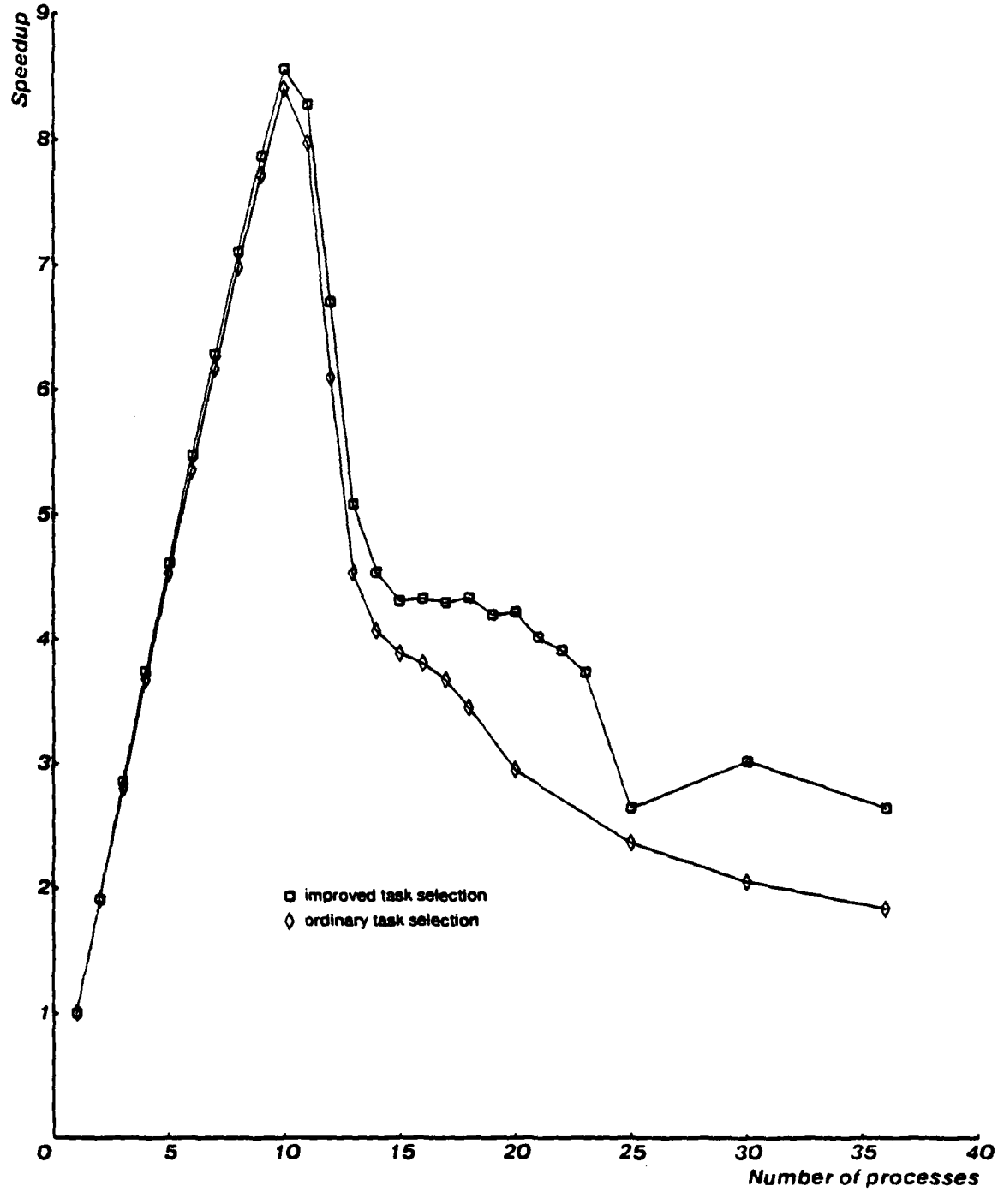


Figure 5-17: PDE—STAROS, Speedup with Different Task Selection

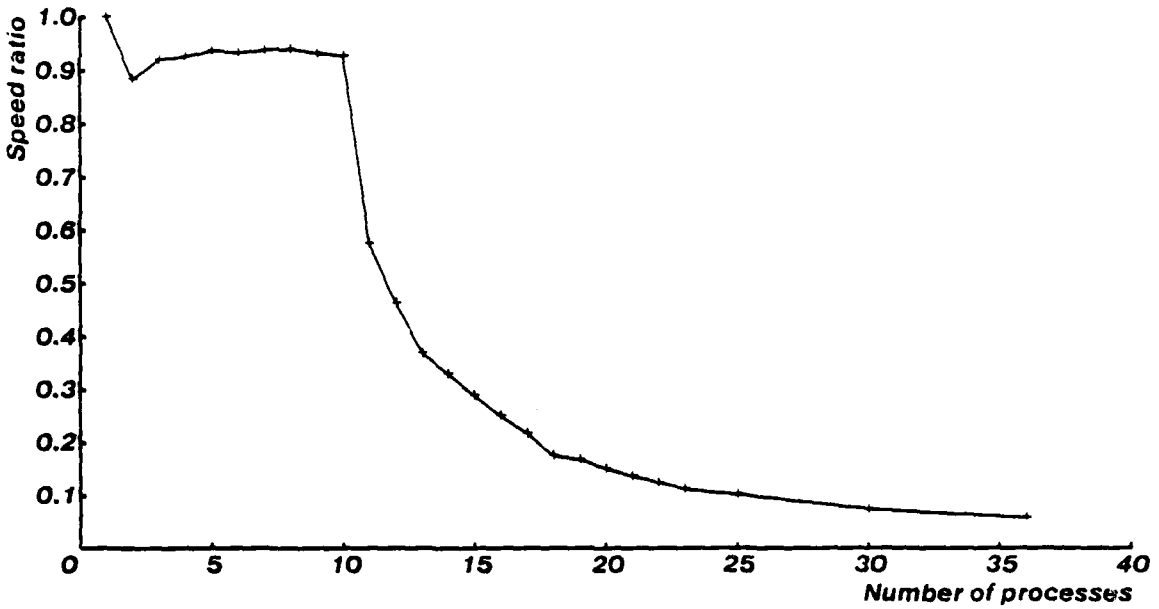


Figure 5-18: PDE—STAROS, Speed Ratio of Different Processes

intercluster memory reference from the slowest process.

Figure 5-19 compares the speedup in two versions of the environment. In this case the speedup in both versions is relative to a uniprocess task force with relocation disabled. In one version the idle processors were halted and performed their serial-line references through the Kmap. Because in the STAROS environment it is not possible to disable relocation during the experiment, idle processors in the second version were running an idle loop in local memory, making no references through the Kmap. Similarly as in the case of SMAP (*cf.* Figure 5-15), there is no significant difference in process performance. This suggests that the STAROS microcode does not saturate the processing power of the Pmap or the bandwidth of the Kbus while serving a unicluster configuration. Contention is the only problem.

Figure 5-20 presents the results with the distributed data for two different versions of the microcode. The *old* version is the version used for all the other experiments. The *new* version was released only recently and has a number of improvements. In particular both intercluster and intracluster memory references have been optimized. Improvement is visible, though less pronounced than with SMAP. With the old version of the microcode, the best speedup is achieved for 18 processes and then starts to drop, probably because of Kmap saturation. For the new version, the

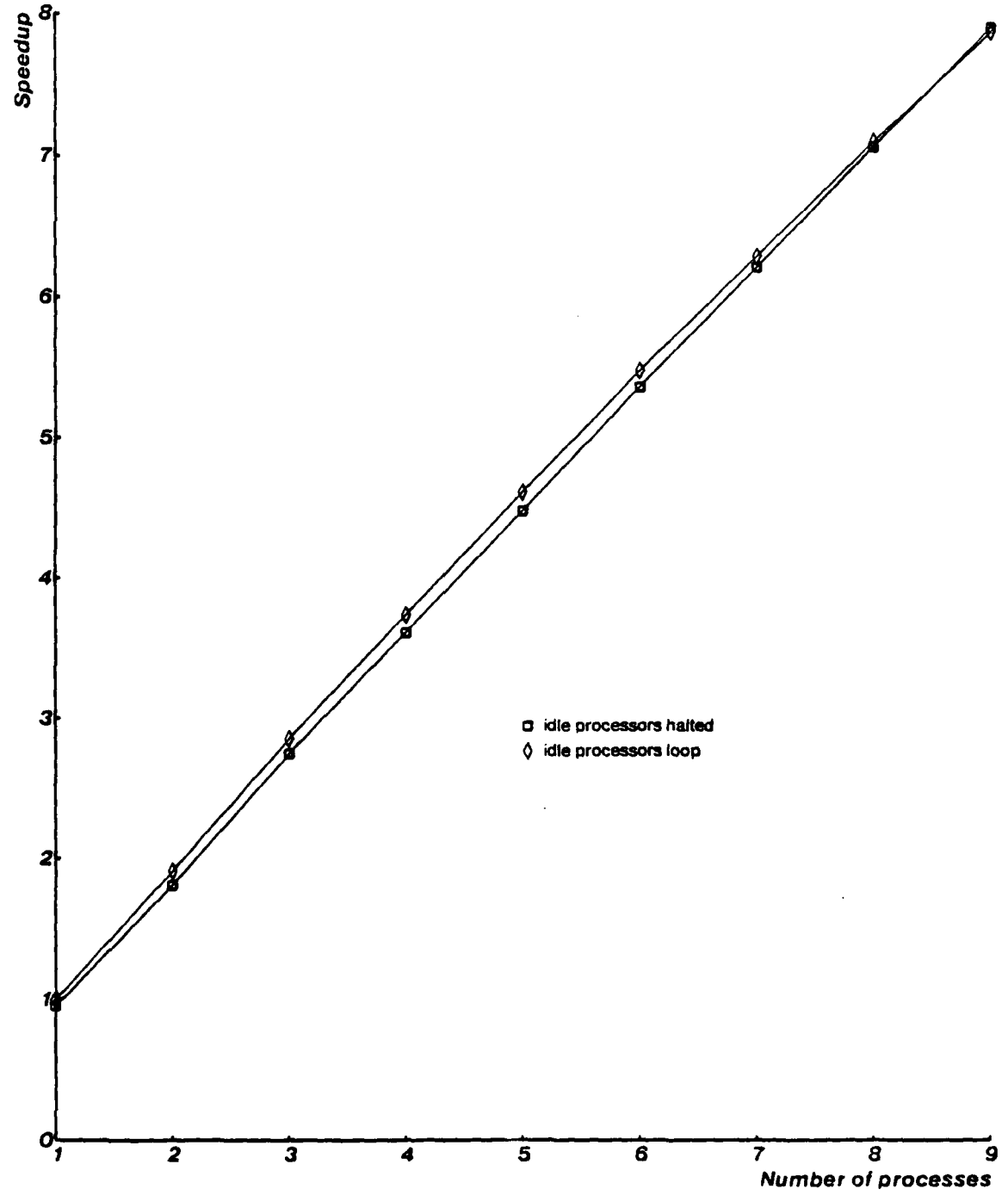


Figure 5-19: PDE—STAROS. Speedup with Different Behavior of Idle Cm's

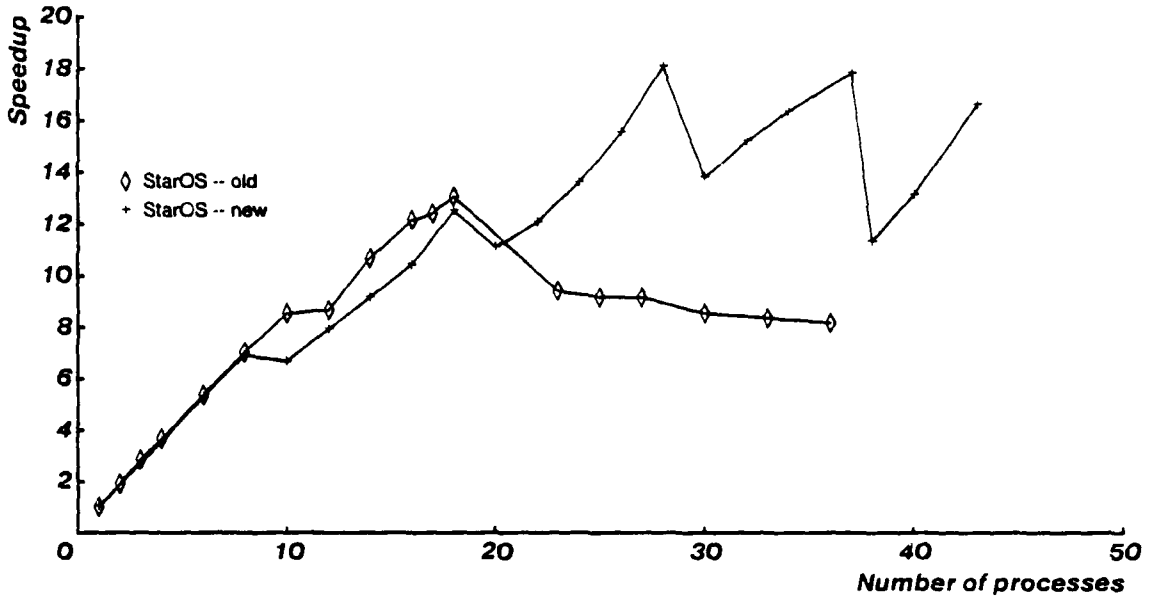


Figure 5-20: PDE—STAROS, Speedup with Distributed Data

speedup grows up to 28 processes, where it reaches the maximum value of 18.1. At that point the slowest process runs with 66% of its potential speed. One may notice that the crossover phenomenon mentioned before causes much more significant performance degradation with STAROS than it does with SMAP. The performance drops significantly each time a part of data is moved to a new cluster with only few processes running in it. That means that the program using STAROS is more sensitive for unoptimal data distribution than the one using SMAP. This is due to the greater difference in time for an intercluster vs. intracluster memory reference in STAROS than in SMAP.

There are ways to improve this performance further. It should be possible to find a better distribution of data between Cm's or between clusters, and perhaps to limit the number of processes running simultaneously in a cluster. Unfortunately, NEST provides only limited tools to reconfigure objects dynamically in Cm*. Ideally, this will be one subject of experiments with the complete STAROS.

Figures 5-21, 5-22, and 5-23 sho v results obtained with the MEDUSA version.

Figure 5-21 shows speedup with all shared data centralized in one Cm. From comparison with Figures 5-9 and 5-17, it may be concluded that the MEDUSA microcode behaves similarly to the

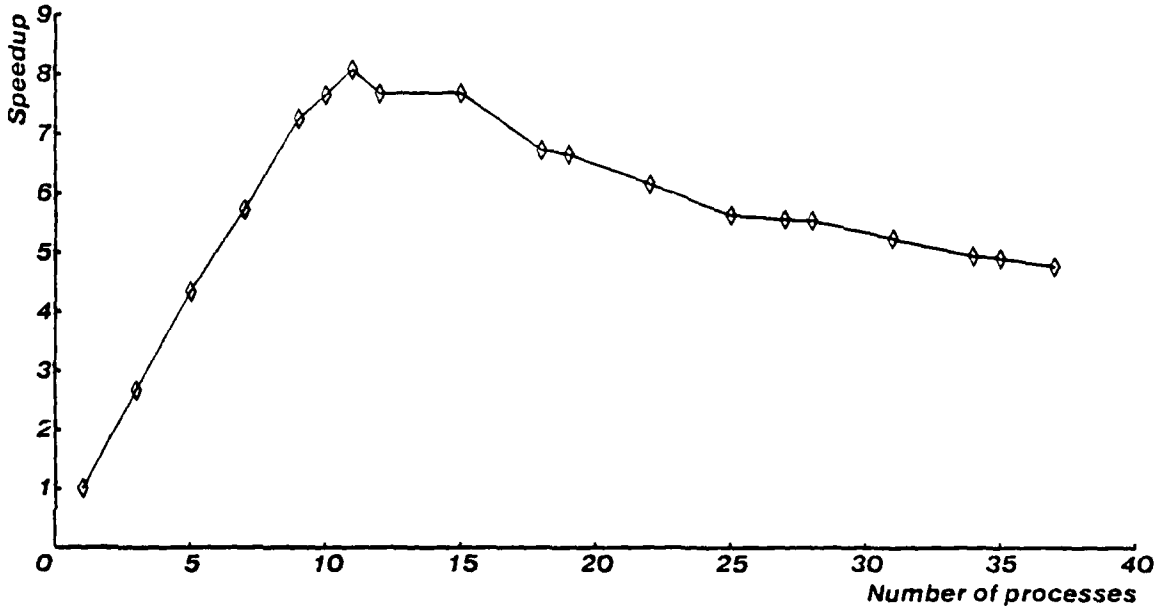


Figure 5-21: PDE—MEDUSA, Speedup with Centralized Data

STAROS microcode, but the performance drop after crossing a cluster boundary is not so large and the results for a large number of processes are significantly better. For more than 35 modules the speedup is still about 5, while with STAROS it has dropped to less than 3. Nevertheless, the maximum speedup is about 8, much less than with SMAP. The reason for that is again, as it was with STAROS, that MEDUSA microcode is more sophisticated than SMAP, and that worse performance is the price for this.

Figure 5-22 presents the speed ratio of the slowest versus the fastest process in a task force with centralized data. As compared with Figures 5-13 and 5-18, this ratio lies between the ones for SMAP and for STAROS. For a large number of processes it is about 15% (85% of time is wasted), while for SMAP this value was 25%, and for STAROS less than 5%.

Finally, Figure 5-23 shows results obtained with data distributed between clusters. Improvement is visible, though not as good as with SMAP. Still, the best speedup is only about 19, while with SMAP it was possible to obtain a speedup of 26 (*cf.* Figure 5-16). Because of the crossover phenomenon, the speedup does not grow continously, but drops each time a cluster boundary is crossed.

Figure 5-24 compares the results of PDE with distributed data for all three microcodes. Only the points where the data distribution is close to optimal are included in this figure, in order that the graph

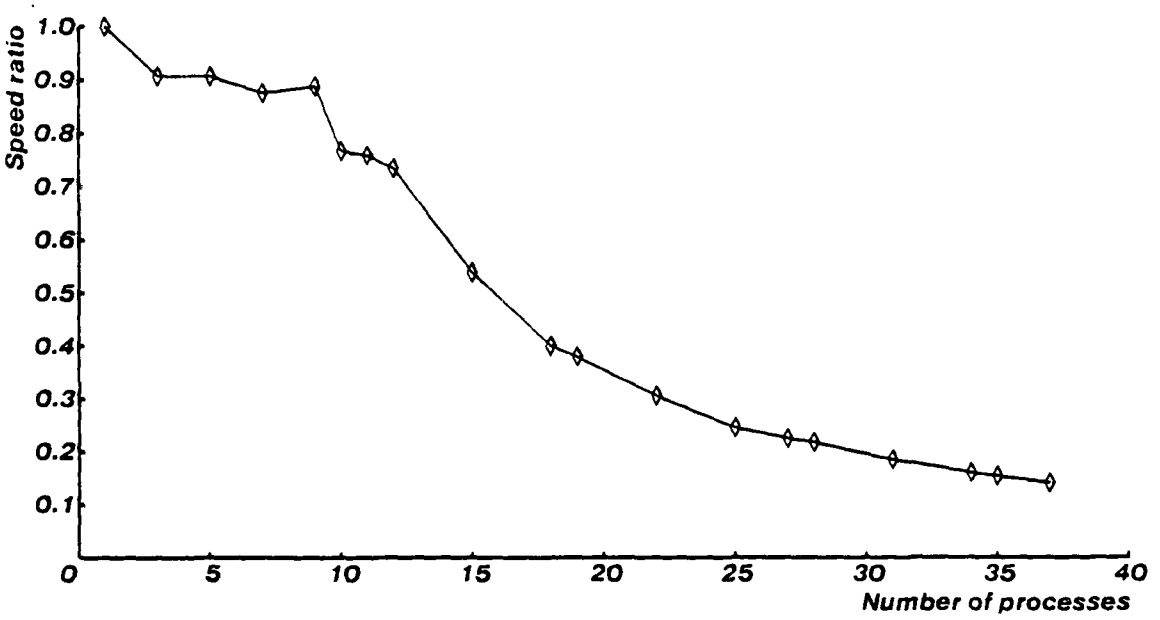


Figure 5-22: PDE—MEDUSA, Speed Ratio of Different Processes

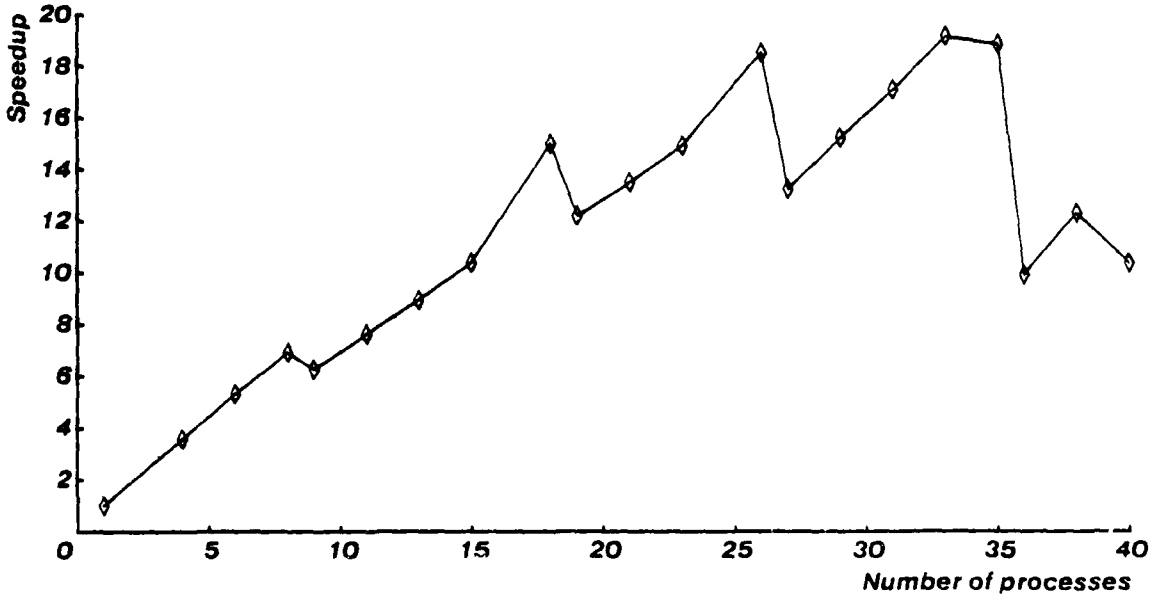


Figure 5-23: PDE—MEDUSA, Speedup with Distributed Data

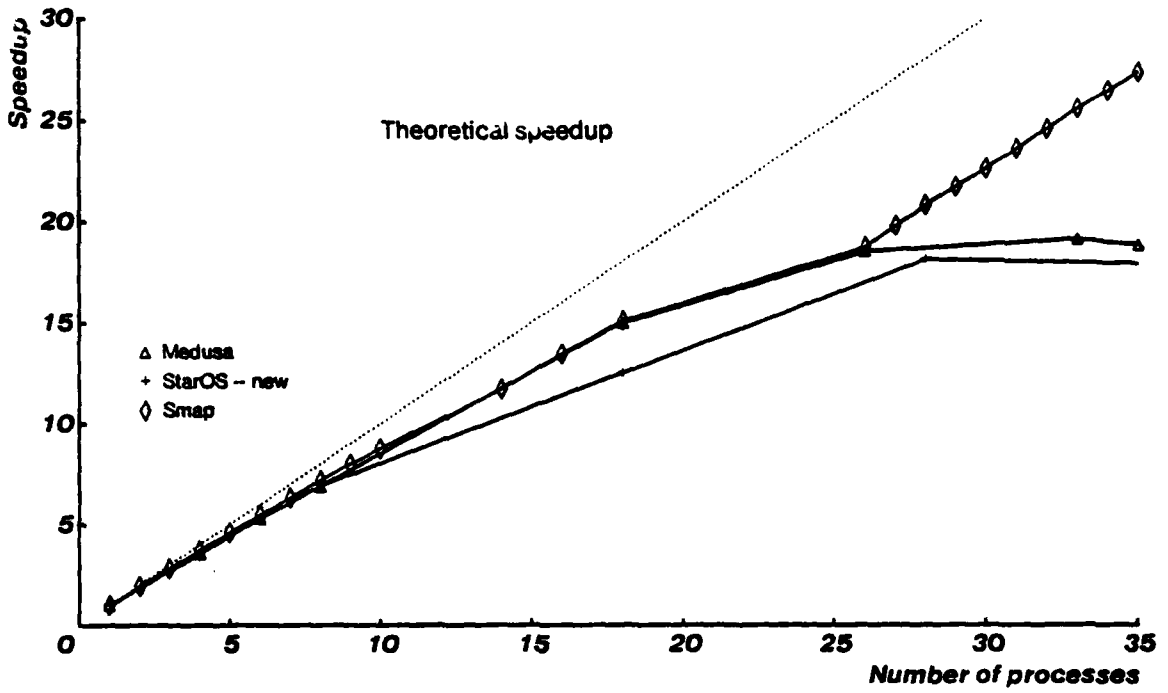


Figure 5-24: PDE with Distributed Data—Comparison

not exhibit the crossover phenomenon. As can be seen, for less than 25 processes (3 clusters), the speedup is close to linear for all three microcodes.

5.4.2.2. Conclusions

The best speedup achieved with the NEST version with non-distributed shared data was 16 (26 Cm's) for SMAP and 8 to 9 (11 Cm's) for both STAROS and MEDUSA microcodes. Respective results with distributed data were 29 for SMAP, and 18 to 19 for STAROS and MEDUSA. For a detailed explanation of differences between the microcodes, see Chapter 10. Those results suggest that PDE may be distributed and run cost-effectively on Cm*. Several minor modifications to the algorithm itself, like the improved task selection, are necessary because of the hierarchical structure of the computer. The reference pattern may raise serious problems of Kmap saturation (in some parts of the algorithm there is one reference to the global data each 4 LSI-11 instructions). A crucial problem concerns the appropriate distribution of the global data between clusters and Cm's based on the (known) reference pattern of particular processes. Experiments directed toward this goal should be conducted as soon as tools become available which are more flexible and more convenient than the

presently existing ones.

As a side effect, the experiments with PDE provided some data on hardware reliability. A special facility in NEST allows a user to run several experiments semi-automatically. In this mode the user records a plan with the attributes of the experiments and NEST runs them one by one and records their results. Using this facility, the program was run several times for more than one day with no restart or other interference from outside. The longest run lasted about three days. No hardware errors were observed during that time. There are no error-correction tools in NEST, so any error would cause halting of the system. In total, the number of errors that occurred during all experiments was very small.

5.4.3. Net

Finally, an attempt was made to run an application with more a complicated reference pattern and more diversified processes. The **net** simulation was created in 1979 and exists only in the NEST version.

In this application, a railway network is simulated by a task force which consists of a fixed number of processes, each of which represents a **station**—one node in the railway net. Two stations may be connected with each other by a unidirectional **track**. For a given station *A* a set of **previous stations** includes each station *B* such that there is a unidirectional track from *B* to *A*. The stations communicate with each other by exchanging messages representing **trains**. The route of each train through the network is an attribute of the train and is determined by some data associated with the train when the train is created. Each station schedules the trains that have arrived according to the time of their arrival and serves them in this order. Each process maintains its own simulated time. At any given moment, this time will probably be different in different processes. Thus, the *simulated* time of sending a train from one station to another is not connected with the *real* time of creating the message representing the train. Also the *real-time* order of those events may be quite different than their *simulated-time* order. Thus, for example, station *A* may send a message to station *C* at real time 5. At this moment the internal clock on station *A* may show the simulated time 50. Station *B* may send a similar message to *C* at real time 7, but its clock may at this moment show the simulated time 40. This second message should be serviced by *C before* the first one, since only the simulated time is relevant. In general, to determine the *simulated-time* order of arrival of the trains, the station must know the *simulated* times of arrival of the next trains from all previous stations. Unless the messages from these stations have been recieved, the station is not ready to run, and it will not be allowed to run by the NEST multiplexer.

The number of processes in this application is fixed and independent of the number of processors.

In the present version there are 63 station-processes. Usually several processes are executed by each processor. The processes are distinguishable, since each of them corresponds to a different station. In addition, they run different programs, as there are two different types of stations: one creates new trains according to the simulation parameters, and the other only schedules arriving trains and sends them further. There also exists one additional process, the **reporter**, which records data from the special messages sent to it by the stations.

This application, as programmed, may not be run on a uniprocessor configuration. The reporter must always be running throughout the experiment. It may not be multiplexed, or otherwise all other processes would be blocked. Moreover, there are too many other processes to put them all into one Cm. In fact, a minimum of 4 Cm's have to be assigned to this task force.

A theoretical estimation of the speedup vs. number of processors is impossible. The reference and communication pattern is very complicated. The time of an experiment depends not only on the number of processors, but also on the distribution of processes between them. At present a process is assigned to only one processor. This application corresponds well to the Cm* structure. Most of shared data is distributed between Cm's. By careful allocation of Cm's to particular processes, a high degree of locality may be obtained.

5.4.3.1. Results

As there is no uniprocessor version to act as a reference point, we cannot compute speedup. Instead, the results will be shown in terms of absolute run time.

The number of experiments conducted with this application was relatively small, as this application is not very suitable to the simple NEST environment. Besides, the parameters such as number of trains, configuration of the simulated net, and number of processes were constant and hard to change.

Figure 5-25 shows the results of experiments with SMAP. The two versions differ in one parameter that determines the travel time of train between stations. It influences the average number of processes blocked at any given time. As can be seen, the time significantly decreases up to approximately 15 Cm's. The very large performance improvement with the small number of processors corresponds to the situation when the average number of runnable processes is greater than the number of available processors. For a larger number of processors, a processor may be idle for some time when all its processes are blocked. Performance of the version with less frequent blocking is better than performance of the second one.

Figure 5-26 shows the results with the STAROS microcode. This graph is much less regular than the previous one. One explanation may be that the relative position of two different processes

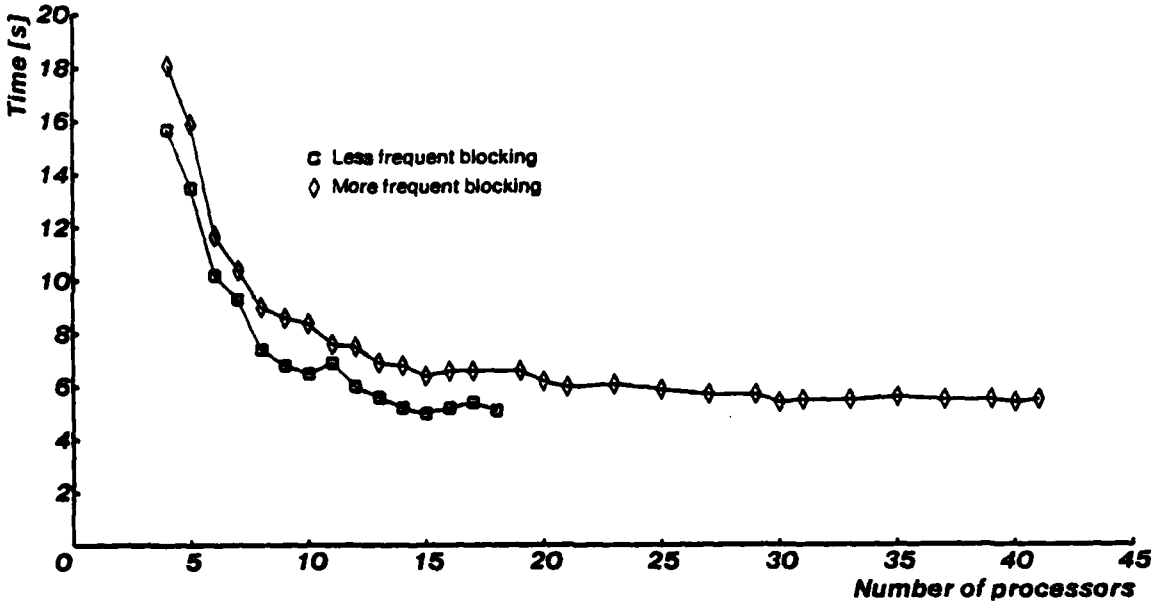


Figure 5-25: NET—SMAP, General Results

communicating with each other may have more influence on performance in STAROS than in SMAP, as in the former, there is greater overhead for intercluster references.

5.4.3.2. Conclusions

From this limited amount of data it seems that this application is relatively well suited for distribution. Its performance could likely be improved further. As the communication between processes is not so trivial as in QSORT and PDE and the process address space changes during an experiment, this application is probably especially suitable to run with the support of an operating system. It will be interesting to compare these results with the results of the application running with the full support of STAROS, possibly using the STAROS message operations for the interchange of information between processes. Also, it will be possible to investigate whether the performance can be significantly improved by careful assignment of processes to processors. Several attempts proved that, with a particular number of processors, it was sometimes possible to obtain up to 20% performance improvement due to better processor utilization. It is not clear whether there exists any general method to do this. In the experiments reported here this assignment was performed in a very straightforward way.

It will be also possible to allow one process to move between processors. This may improve

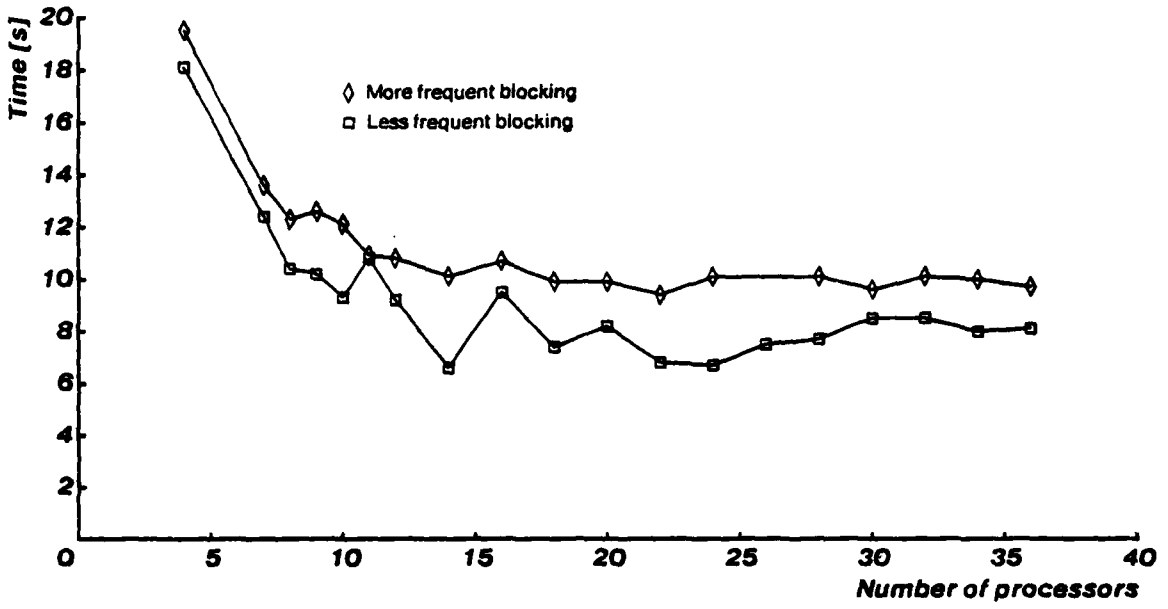


Figure 5-26: NET—STAROS, General Results

processor utilization, but will increase overhead associated with references to the process stack and local data (if they are in a module other than the processor running the process).

5.5. References

[Fuller *et al.* 77] S. H. Fuller, A. K. Jones, I. Durham, eds.
Cm review.*
Technical Report, Computer Science Department, Carnegie-Mellon University, June, 1977.

[Raskin 78] Levy Raskin.
Performance evaluation of multiple processor systems.
PhD thesis, Carnegie-Mellon University, August, 1978.
Available as CMU tech report CMU-CS-78-141.

[Sedgewick 78] R. Sedgewick.
Implementing quicksort programs.
Communications of the ACM 21(10):847 – 57, October, 1978.

6. Applications Programs on Cm*

Most of the effort in programming Cm* has thus far been directed toward developing system software. Serious attempts to use Cm* for research in various scientific disciplines are just beginning. Nonetheless, preliminary results are available for three experiments. The first section of this chapter describes the simulation of an electrical power network. The next section reports on two projects to apply the computational power of Cm* to problems in chemistry which are amenable to multiprocessor solution. The final section describes the use of parallel processing to remove hidden lines in the portrayal of three-dimensional objects on a graphics display.

6.1. Power System Simulation on Cm* *Ivor Durham*

The task of simulating large electrical networks is the subject of extensive work in the Power Systems Industry. One algorithm currently in use, the **Network Model**, exhibits inherent parallelism in stages which account for over 50% of its time [Dugan 79]. The goal of this work was to develop an experimental simulator on Cm* with which to investigate the potential parallelism in this algorithm.

The description given below is a condensed and edited version of the paper on this work presented to the IEEE Power Engineering Society Summer Meeting in Vancouver in July 1979 [Durham 79].

6.1.1. The Network Model and Simulation Algorithm

The Network Model is described here informally. A mathematical and more detailed description can be found in [Dugan 79, Talukdar 76]. The model upon which the algorithm is based is analagous to the data-type model for software engineering: An electrical network is composed of a set of interconnected primitive components. An interconnected subset of the devices may be considered as a **complex** device and handled in the same way as more primitive devices. Any device (primitive or complex) may be characterized by the behavior of the voltage and current values at its terminals. The mathematical model for a complex device (or sub-network) is termed a **macromodel**. The macromodel may range in complexity from the single equation embodying Ohm's law for a two-terminal resistor to hundreds of differential equations representing multi-terminal devices like transmission lines.

The overall behavior of a network at any instant in time is determined by solving the linear equations embodying Kirchoff's constraints for the macromodel terminals together with the equations contained in the macromodels. The solution proceeds iteratively. The collective solution of the

macromodels is referred to as **Phase I** of a simultation step and the part devoted to the solution of Kirchoff's constraints as **Phase II**. A number of iterations through the two-phase series may be required to converge the results before proceeding to the next simulated time step.

There is inherent parallelism in this model because the macromodels describing the devices may be solved independently. Experience with a uniprocessor implementation of this algorithm [Talukdar 76] has shown that, depending on the number of devices in the network, over 50% (and sometimes as much as 97%) of the computation time is spent solving the macromodels.

6.1.2. The Problem Decomposition

The decomposition of the simulation algorithm for Cm* was guided by two main criteria:

- The system must support experimentation. An important part of this work is to identify in as much detail as possible where the computing power can be and is being used. For example, we would like to know how much time is spent in synchronizing between processes, how much time is spent in administration of the shared data structures, and how much time is devoted to the real processing of the model.

- A considerable amount of effort has been invested elsewhere in exploiting parallelism to solve linear systems [EPRI 77, Wallach 74a, Wallach 74b]. The simulation system for Cm* should attempt to exploit the parallelism inherent in solving macromodels.

For any network, one macromodel may describe a number of similar (complex) devices. There may be several transmission lines, for example, each characterized by different parameter values. All devices described by the same macromodel are collected into a **device pool**. An arbitrary number of processes (processors) may be applied to the solution of the macromodel for a particular pool of devices. The selection of the number of processes is based on the complexity of the macromodel and the number of devices in the pool.

During an iteration, each Phase I process repeatedly extracts an unprocessed device from its associated device pool. The new voltage and current values for the device's output terminals are computed from the corresponding values for its input terminals according to the macromodel. The resulting terminal voltages and currents are contributed to the linear system that described Kirchoff's constraints.

When all of the devices in all of the device pools have been processed, the Phase II computation may be performed to solve the linear system. Information characterizing the state of the network, such as node voltages, may be reported at the end of Phase II. (In the implemented system, these intermediate results are shipped to another computer for analysis.)

6.1.3. The Implementation

The implementaton of the prototype system is described here in four parts:

- Control structure and general organization
- Macromodels and device pools
- Phase II
- User interface

The first version of the StarOS operating system [*Jones et al. 78*] (*see Chapter 7 for a description* of the current StarOS system) provided the program environment for the simulator. The simulator ran on a single 10-processor cluster,

Control Structure and General Organization. The activity of the simulator is coordinated by a **management process.** A set of Phase I processes are responsible for solving the macromodels for the devices they extract from the device pools. The single Phase II process waits for Phase I to complete simply by watching a decrementing count of the devices processed. Hence, the locus of control moves around a cycle from the management process to the Phase I processes (in cooperation) and finally on to the Phase II process.

The Phase I processes simply wait until there are unprocessed devices in their particular device pool. The management process puts the devices back into the pools to start a new iteration. In addition, it resets the decrementing counter of processed devices to alert Phase II that a new iteration has started. Since Phase II watches this counter, the management process is not involved in starting the Phase II processing. Phase II notifies the management process (via a shared variable) when it has completed. The management process may choose to ship intermediate results to another computer concurrently with the next iteration.

Macromodels and Device Pools. A particular device is represented by its characteristic parameters for the macromodel that simulates its behavior. In the implementation, each device's parameters and terminal voltage and current values are represented in a StarOS basic object (referred to as a **device segment**). A device pool is simply a set of device segments. All of the Phase I processes that are to work on a particular macromodel access the device pool through shared memory.

The device pool is represented as a simple vector of device segments with a counting lock. The management process "puts" the devices into the pool by resetting the counting-lock value to the number of devices in the pool. The Phase I processes identify the next device to process by using the result of the StarOS indivisible decrement operation on the counting lock. The result is the index of a capability for an object containing the device segment. Hence, while the index is zero, there is no

device to process. After the management process resets the counting lock, the resulting index is non-zero as long as there are devices left to process.

After selecting a device from the pool, the Phase I processes solve their macromodel using parameters from the acquired device segment. The results of this computation are the new voltage and current values at the device's terminals. These values must be contributed to the linear system to be solved in Phase II.

Phase II. The responsibility of the Phase II Process is to solve the system of linear equations that describe Kirchoff's constraints for the network. The linear system is represented as a matrix of coefficients and a vector of node voltages and currents. Access to the shared matrix must be controlled because the contributions by Phase I processes are cumulative rather than absolute. Each Phase I process must set a lock before making any changes.

If no changes are made to the shared matrix during Phase I, there is no need to repeat the factorization of the linear system. Phase II avoids the extra processing by maintaining a second matrix which holds the results of a Gaussian elimination on the shared matrix. The factorization of the matrix represents a significant portion of the Phase II processing.

User Interface. To make the simulator available for experimentation in a comfortable environment, the user interface was implemented on a more mature system—a PDP-10. The **user interface** is an interactive program that accepts the user's description of the network to be simulated and passes it over a communication line to the simulator on Cm*. Conversely, the simulator ships results back to the user interface. The user may then avail himself of the existing software systems on the PDP-10 to analyze and display the results of the simulation.

To achieve the communication with the PDP-10, a **communication process** was provided in the simulator. This process was responsible for the reliable exchange of information between the simulator's management process and the user interface program.

6.1.4. Uniprocessor vs. Multiprocessor Comparison

The design criteria for the experimental simulator ranked flexibility over performance. It was believed that promising versions of the simulator could be optimized for more detailed evaluation. However, some first-order comparisons can be made with an implementation of the same algorithm on a DECSystem-20.

The sample network that was simulated for the comparision consisted of a single-phase, 100 pi-section transmission line [Dugan 79]. Variation in Phase I is achieved by partitioning the pi-sections

to form macromodels of varying computational demands. The Cm* experiments were run on a 9-Cm cluster. The management and communication processes were given dedicated processors. A third processor executed Phase II and provided Phase I service for the device pool containing the single voltage source required in the sample network. Various subsets of the 6 remaining Cm's were applied to the transmission-line device pool during Phase I.

The simulator outputs intermediate results at the end of each time step rather than at the end of each iteration. The experiments were performed by executing in excess of 300 iterations for each of 10 time steps to avoid the performance degradation introduced by shipping the intermediate results (concurrently with Phase I) to the user interface over a low-bandwidth communication line. Each device process had a local copy of its code and private data. Hence all code and stack references were made to local memory.

The predicted time for an iteration on Cm* is described by the following equation:

$$T = T_{\text{Phase I}}/N + T_{\text{Phase II}} + T_{\text{Overhead}}$$

where $T_{\text{Phase I}}$ is the time to perform Phase I with a single processor, and N is the number of processors. For Cm*, the first term is a special case of the more general formula:

$$T = \text{Max}(T_{\text{Macromodel}}) + T_{\text{Phase II}} + T_{\text{Overhead}}$$

The uniprocessor-equivalent time was determined by permitting only one processor to work on the transmission-line device pool during Phase I. Although Phase II was executed on a dedicated processor, it could not begin until Phase I had completed so the resulting performance was equivalent to a completely serial computation.

The Speed-Up factor is computed from

$$S = T_s^n/T_p^n$$

where

T_s^n = Iteration time with a single processor solving the transmission-line macromodel during Phase I.

T_p^n = Iteration time with N processors solving the transmission-line macromodel during Phase I.

Figure 6-1 compares the predicted performance with the measured performance on Cm*. The actual performance of the simulator is substantially different from that predicted by the DECSystem-10 results [Dugan 79]. These results are a dramatic illustration of a typical failing of simplisitic prediction models for multiprocessors: in particular, simplistic models often ignore salient features of the target multiprocessor architecture and overhead introduced by communication and synchronization. The model used in [Dugan 79] assumed a uniform cost for accesses to all memory in Cm*. Making non-local memory references in Cm* incurs a factor of 3 overhead over local references.

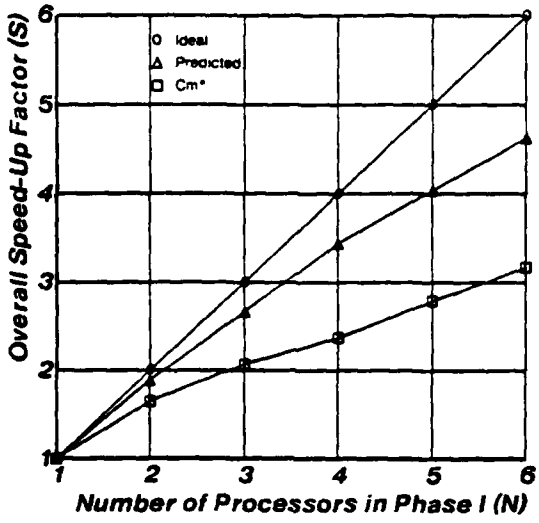


Figure 6-1: Comparison of Predictions and Measurements

Although the simulator guaranteed that all of the processes had a local copy of their code and private data, almost none of the data references were to local memory. The most critical instance of this is the shared matrix which is used by Phase II. Much of the Phase II computation operates in this matrix and *all* references to it are non-local. This affected the results presented above because the times represent entire iterations rather than just Phase I.

If I/O costs are ignored in both the uni- and multiprocessor versions of the simulator, Cm* can execute about 0.93 iterations per second with only one processor working during Phase I. In comparison, the DECSystem-20 can execute about 4.36 iterations per second. When six processors are applied to Phase I, the iteration rate increases to about 3.0 per second. With only one processor working on Phase I, the Cm* simulator is about five times slower than the DECSystem-20 implementation. Further taking into account the slow floating point operations on the Cm* LSI-11 [DEC 75], the Cm* simulation system performs quite well. Given that the raw power of a DECSystem-20 (about 2 MIPS) is close to that of a full (14-Cm) cluster on Cm*, it is evident that modest increments in the problem size will lead to a superior performance by Cm*.

6.1.5. Status

Mike Carey

This work was done during the 1978 – 79 academic year. Research is currently being pursued to investigate further the effectiveness of using multiprocessors for running transient simulations of power systems. This research involves the implementation on Cm* of a small parallel multicluster version of the METAP transient simulation program [Talukdar 76]. Basically, the new implementation is similar to that described above, but it is based on NEST (see Section 5.3.2) and written to handle a small but representative class of power systems. Performance measurements will be taken to determine both the gains afforded by the parallel implementation and the factors limiting these gains. Future work may include research into algorithms which exhibit even greater parallelism or which lend themselves to asynchronous iterative implementations.

6.2. Applications Related to Chemistry

Neil Ostlund and Ed Gehringer

In this section, we delineate two specific applications in the field of chemistry which are amenable to solution by a Cm*-like multiprocessor: molecular orbital (MO) calculations of the electronic structure of molecules, and Monte Carlo calculations of the statistical mechanics of condensed media. These algorithms have been chosen because they do not appear to decompose effectively with only a single instruction stream.

In contrast to the Cm* results, most of the experimental work to date on the parallel decomposition of numerical algorithms has been with single-instruction, multiple-data (SIMD) stream machines such as the ILLIAC-IV or the Cray-1. These architectures are well suited to the solution of sets of linear equations or several more sophisticated, but still relatively straightforward, matrix problems. However, it appears that both of the problems to be detailed in this section have a high degree of parallelism on multiple-instruction, multiple-data (MIMD) stream machines such as Cm*, but little on SIMD machines.

Because the processing power of Cm* is obtained from LSI-11's, our results will have to be scaled to reflect two limitations of the processors, compared to other hypothetical MIMD machines. First, the floating-point abilities of LSI-11's are poor compared to those available on other processors. For example, the cost of a 32-bit floating-point multiply is about 50 μsec. (performed in LSI-11 microcode), and the cost of a 64-bit floating-point multiply is about 3000 μsec. (performed in software). The calculations described here require more precision than is available with a 32-bit floating-point word. Thus Cm* is a 10 Mips machine, but only a .02 Megaflops machine for 64-bit floating-point numbers. Second, the address space is 32K words, and for a processor to access more memory than this requires the loading of relocation registers with a consequent degradation in performance. It should

be pointed out that these factors are a consequence of the implementation of Cm* using LSI-11's, rather than an inherent property of the architecture.

6.2.1. Monte Carlo Calculations Using the Metropolis Technique

The fundamental problem of statistical mechanics is the relation between the macroscopic behavior of matter and the microscopic properties of its constituent parts. We have chosen a particular problem in statistical mechanics: a Monte Carlo study, using the Metropolis technique [Metropolis *et al.* 53], of 256 molecules of water under periodic boundary conditions. The algorithm obtains the properties of a macroscopic liquid by averaging over a large number of random microscopic configurations of a collection of individual molecules. There has recently been a flurry of interest [Pangli *et al.* 78] in developing algorithms closely related to the original Metropolis algorithm and in investigating the advantages of Monte Carlo methods compared to those of molecular dynamics [Stillinger 75]. It will be possible to compare the multiprocessor results directly with extensive related calculations on conventional processors. This problem has a non-trivial computational complexity, and provides an excellent vehicle for studying memory organization, communication, and synchronization of parallel processes.

There have been a number of liquids studied by Monte Carlo simulation, but the most intense recent efforts revolve around the simulation of liquid water [Pangli *et al.* 78] using either an empirical [Stillinger and Rahman 74] or an *a priori* [Matsuoka *et al.* 76] water dimer potential. Thus a number of published studies will be available for comparison with our results.

The Parallel Decomposition Method. The Metropolis algorithm considers a collection of N molecules and generates a sequence of appropriately weighted spatial configurations of these N molecules by a Monte Carlo procedure. Macroscopic quantities are obtained by a simple averaging over a sequence of approximately 10^6 configurations of the system. To generate each new configuration, a single molecule is moved. At each of these configurations, the bottleneck in the calculation involves the evaluation of the $O(N)$ altered intermolecular interactions. The present version of the parallel algorithm uses K processors to evaluate the new $N-1$ intermolecular interactions with a moved molecule. If linear speedup can be obtained, then the complexity of the calculation at each configuration is reduced from $O(N)$ to $O(N/K)$. The algorithm is thus potentially capable of speedup which is linear in the number of processors available. However, to obtain linear speedup requires that memory contention and interprocess bus contention are small, and that synchronization and latency costs are negligible. Initial experiments show that this is far from true.

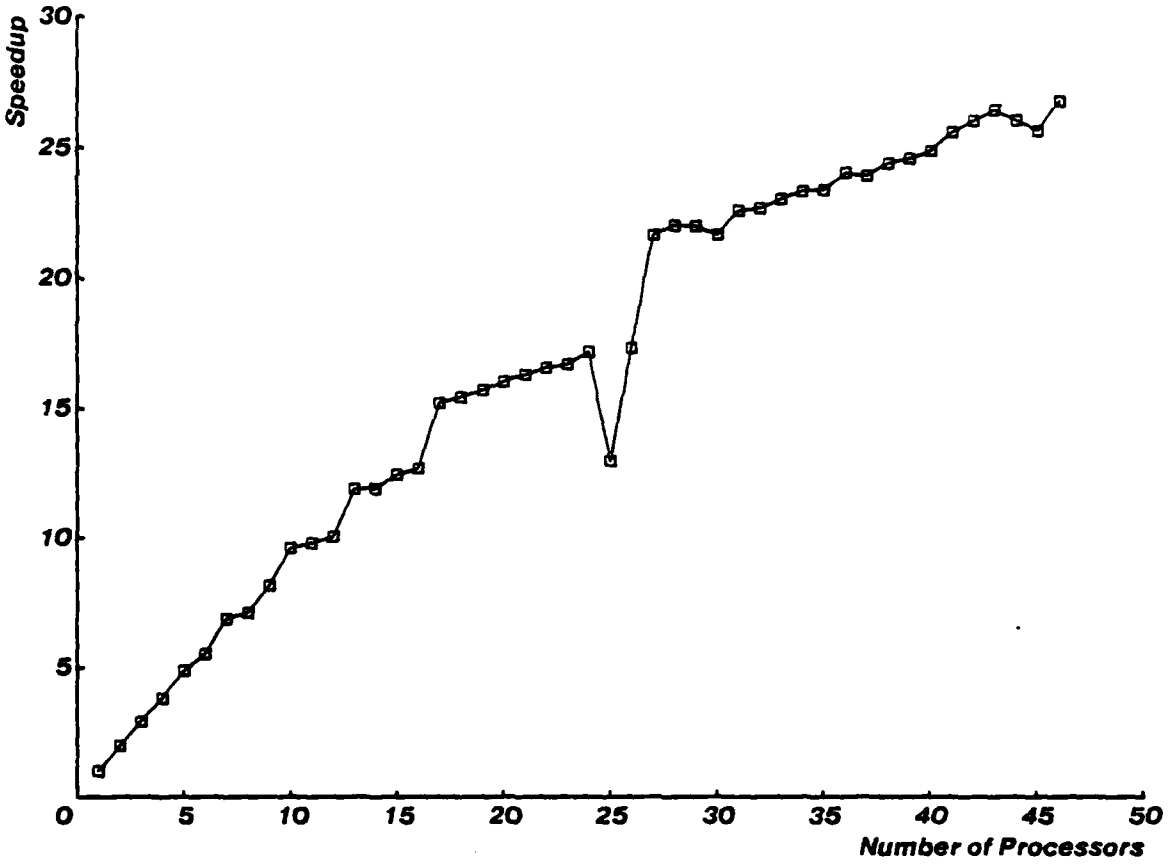


Figure 6-2: Speedup of the Metropolis Algorithm

Preliminary Results. A number of results using Cm* have been obtained for versions of this parallel algorithm. The present results are limited, however, to a small number of particles and to an interparticle potential appropriate only to atoms rather than molecules. Figure 6-2 displays the results obtained with a system of 50 atoms[11] [Rossky *et al.* 78]. The maximum speedup—nearly 27—was observed with 46 processors. This relatively optimistic result, however, does not take into account the fact that floating-point operations were performed in software rather than hardware. The major communication and synchronization bottleneck in these calculations involves the global addition of individual interaction terms evaluated by different processors. It is expected that the algorithm will

[11] On two separate occasions during the preliminary experiments, we noted an anomaly in the curve, like the one shown in the graph at 25 processors. We have not yet isolated the cause.

improve as the number of particles increases relative to the number of processors. For this more important case, most of the additions can be performed locally.

While reasonable experience with parallel Monte Carlo algorithms running on Cm* is now available, there has not been an effort to run, from beginning to end, a truly significant scientific problem. This will be the next step in our investigation. In particular, the present limited programs will be extended from using a spherical potential to using the general non-spherical intermolecular potential, and from treating only a simple cluster of particles to using the usual periodic boundary conditions necessary to represent an infinite system. Calculations of the structure of liquid water, as described above, will then be investigated.

6.2.2. Molecular Orbital Calculations

Molecular orbital calculations constitute another "real-life" application whose solution has wide-ranging consequences. In addition, the techniques used to solve these problems have sufficient diversity that they provide a thorough test of the many aspects of Cm*. These calculations are of great interest to a large section of the scientific community and consume a major fraction of academic computing resources. The MO algorithm in its serial form has been extensively developed by generations of quantum physicists and chemists. It includes many *matrix manipulations* (diagonalization, inversion, etc.), *an iterative self-consistent procedure*, extensive integral evaluations, and large data-handling problems. The bottleneck in MO calculations is the evaluation and manipulation of the order of N^4 integrals. These integrals fall into a very large number of different types, and the degree of branching in the related code is such that only a MIMD architecture like Cm* can provide maximum speedup.

There are two places in the MO algorithm where it is necessary to perform $O(N^4)$ operations: initially, a set of two-electron integrals must be evaluated, and later $O(N^4)$ operations are required to form the *Fock* matrix from the two-electron integrals at each step of the iteration. The evaluation of the $O(N^4)$ two-electron integrals at the beginning of each calculation can be easily formulated as a set of disjoint processes.

Many of the other steps involved in the molecular orbital calculation are simple matrix manipulations. In particular, the $O(N^4)$ steps involved in forming the Fock matrix are essentially a square-matrix, column-matrix product and thus an inherently parallel task. Individual iterations can therefore be expected to enjoy few communication and synchronization costs. If the iterations were to proceed in a lock-step synchronized fashion, the overheads here might limit speedup. Recent results [Baudet 78, Raskin 78] suggest, however, that these costs can be reduced substantially by performing the iterations in an asynchronous fashion. The molecular orbital calculation, indeed,

provides a good tool for experimenting with synchronous versus asynchronous iterations. It appears that speedup in molecular orbital calculations will not be substantially limited by serial processes and synchronization overheads.

The second factor affecting speedup is the hit ratio, which is the ratio of local memory references to total memory references. This factor is unique to the Cm* switching structure. The penalty paid for memory references to another computer module or another cluster necessitate a high hit ratio for effective performance. Since 70 – 80% of all memory refences are to code, this requires that the code be duplicated in each computer module. Previous results [Swan 78] suggest that there is about a 17% degradation in the performance for a 90% hit ratio, and only a 2% degradation for a 99% hit ratio. We indicate below that one can expect a Cm* parallel implementation of the molecular orbital algorithm to achieve a high hit ratio.

Solution of the molecular orbital equations involves many matrix manipulations. If one partitions an $N \times N$ matrix into blocks according to the number of processors K, with each processor's memory holding a block, then one can show that ordinary square matrix multiplication involves $N/K^{1/2}$ local operations per global fetch, where an operation here is defined as a floating-point multiplication. For 100 x 100 matrices on Cm*, this reduces to fourteen floating-point manipulations per global fetch, when all 50 processors are in use. With local code, one might therefore expect a hit ratio in excess of 95% for these manipulations.

The formation of the Fock matrix each iteration involves N^2/K floating-point operation for every fetch of a global density-matrix element. For N = 100 with 50 processors, this reduces to 200 floating-point multiplications per global fetch. With local code and local two-electron integrals, one thus expects a hit ratio of greater than 99% for this bottleneck step.

6.3. Hidden-Line Elimination on Cm* *Satish Gupta and Bob Sproull*

Hidden-line elimination for three-dimensional computer images is another experiment in the study of Cm* as a useful and viable computing structure. The following paragraph from [Newman and Sproull 79] describes the task.

> "One of the more challenging problems in computer graphics is the removal of hidden parts from images of solid objects. In real life, the opaque material of these objects obstructs the light rays from hidden parts and prevents us from seeing them. In the computer generation of an image, no such automatic elimination takes place when objects are projected on the screen coordinate system. Instead, all parts of every object, including many parts that should be invisible, are displayed. In order to remove these parts to create a more realistic image, we must apply a *hidden-line* or *hidden-surface algorithm* to the set of objects."

We wanted to study parallel algorithms for this problem. We wanted to see if one should adapt and use different decompositions of the algorithms as the statistics of the image change.

6.3.1. Experiment Performed

The algorithm chosen for implementation was Warnock's hidden-line elimination algorithm. This algorithm was chosen mainly because it seemed easy to implement and also because we used a line-drawing display which is well suited to display the output of this algorithm. Newman and Sproull [Newman and Sproull 79] describe the algorithm.

> "Warnock developed one of the first area algorithms, which selects windows by a recursive procedure. The algorithm first tries to 'solve' the hidden-surface problem for a window that covers the entire screen. If polygons overlap the window in *x* or *y*, a decision procedure is invoked that tries to analyze the relationships between the polygons and generate a display for the window. Simple cases, such as one polygon in the window or none at all, are easily solved.
>
> "If a window is too complicated for the decision procedure to display directly, the algorithm divides the window into four smaller windows and recurs, processing each one with the same algorithm. This technique gives rise to a tree of window subdivisions. If a region of the image is very complex, the recursion will force analysis of smaller and smaller windows. The recursion terminates either by eventually finding a window that can be solved or by finding a window that is as small as a single pixel on the screen. In this case, the intensity of the pixel is chosen to represent one of the polygons visible in the pixel."

The most obvious parallel implementation is used. A global stack of unprocessed windows is maintained. Processors are constantly looking at the global stack, and when they find it non-empty, they pop one of the windows and apply the decision procedure on that window. At the termination of the procedure one of two things happens. Either the window was too complicated, in which case the processor pushes four more sub-windows on the window stack. Alternatively the window was simple, in which case the processor has solved the problem for that window which will result in displaying the contents of the window on the display. Figure 6-3 gives an abstract description of the outer loop executed by all the processors.

We use a Graphics Display Processor [Rosen 73, Rosen 74] to display the images. The GDP is made to emulate a Tektronix 4000 graphics display. It is connected directly to one of the Cm's in Cm* by a 4800-baud Asynchronous Serial Line Interface. This Cm also serves as a master and controller for the experiment.

The experiment is started by loading and starting up the master Cm. It queries the user about the configuration of the experiment. The user has to specify the number of processors to be used, and the clusters in which they reside. The master Cm will then load and initiate the computation on the selected Cm's. The master Cm also maintains the global data structures such as the window stack,

```
{ The following defines a record type for square windows.
  Left and Bottom specify the position of the left bottom
  corner while Size gives the length of a side. }

      Window = RECORD ( Left,Bottom,Size : INTEGER )


{ The following declares a stack of windows and initializes it
  with one window which is the whole screen. }

      WindowStack : GLOBAL STACK OF Window
              INITIALLY [(0,0,1024)]


{ This is the outer loop executed by all the processors.
  Warnock is the decision procedure. }

      WHILE True DO BEGIN
          IF NotEmpty(WindowStack) THEN
              Warnock(Pop(WindowStack))
          END
```

Figure 6-3: Outer Loop of Warnock Algorithm

the output queue and other status information.

The test data is a simple scene consisting of six squares which are stacked so that all except the one on the top are partly obscured. The execution time to solve this problem is 53.5 secs. using one processor, 27 secs. using two, and 6.2 secs. using ten processors. Initially, we observe an almost linear speedup as the number of processors is increased. The speedup becomes sublinear as the number of processors is increased further. The number of processors that can be used while still providing almost linear speedup depends upon the number of polygons in the scene and the complexity involved. But for reasonable problem sizes we expect 50 processors to provide a linear speedup in the time of computation.

6.3.2. Planned Work

Running standalone applications on Cm* is hard because of two major problems. First of all, most of the programming effort goes into providing facilities that are conventionally provided by an operating system. Secondly, it is extremely hard to get programs loaded into Cm*, because the existing paths from a good programming environment are extremely slow and tedious. So it was decided to postpone the rest of the experimentation till one of the operating systems is usable. My

current plan is to investigate adaptations of scan-line algorithms on Cm*. Scan-line algorithms solve the hidden-surface problem by processing the image one scan line at a time. These algorithms could be decomposed by assigning different sets of scan lines to different processors.

6.4. References

[Baudet 78] G. M. Baudet.
The design and analysis of algorithms for asynchronous multiprocessors.
PhD thesis, Carnegie-Mellon University, April, 1978.

[DEC 75] Digital Equipment Corporation.
LSI-11 Processor Handbook
Maynard, MA, 1975.

[Dugan 79] R.C. Dugan, I. Durham, and S.N. Talukdar.
An algorithm for power system simulation by parallel processing.
In *Text of abstracts, Summer Power Meeting*. IEEE Power Engineering Society, 1979.

[Durham 79] I. Durham, R.C. Dugan, A.K. Jones, and S.N. Talukdar.
Power system simulation on a multiprocessor.
In *Text of abstracts, Summer Power Meeting*. IEEE Power Engineering Society, 1979.

[EPRI 77] Electric Power Research Institute.
Exploring applications of parallel Processing to power system analysis problems.

[Jones *et al.* 78] A. K. Jones, R. J. Chansler, Jr., I. Durham, P. Feiler, D. A. Scelza, K. Schwans, and S. R. Vegdahl.
Programming issues raised by a multiprocessor.
Proceedings of the IEEE 66(2):229 – 37, February, 1978.

[Matsuoka *et al.* 76]
O. Matsuoka, E. Clementi, and M. Yoshimine.
CI study of the water dimer potential surface.
J. Chem. Phys. 64:1351 – 61, 1976.

[Metropolis *et al.* 53]
N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller.
Equation of state calculations by fast computing machines.
J. Chem. Phys. 21:1087 – 92, 1953.

[Newman and Sproull 79]
W. M. Newman and R. F. Sproull.
Principles of Interactive Computer Graphics, 2nd edition.
McGraw-Hill, 1979.

[Pangli *et al.* 78] C. Pangli, M. Rao and B. J. Berne.
On a novel Monte Carlo scheme for simulating water and aqueous solutions.
Chem. Phys. Lett. 55:413 – 17, 1978.

[Raskin 78] Levy Raskin.
Performance evaluation of multiple processor systems.
PhD thesis, Carnegie-Mellon University, August, 1978.
Available as CMU tech report CMU-CS-78-141.

[Rosen 73] Brian Rosen.
The architecture of a high-performance graphics display terminal.
SID International Symposium Digest , May, 1973.

[Rosen 74] Brian Rosen.
Graphics display processor programmers guide.
Technical Report, Department of Computer Science, Carnegie-Mellon University, 1974.

[Rossky *et al.* 78] P. J. Rossky, F. D. Doll, and H. L. Friedman.
Brownian dynamics as smart Monte Carlo simulation.
J. Chem. Phys. 69:4628 – 33, 1978.

[Stillinger and Rahman 74]
F. H. Stillinger and A. Rahman.
Improved simulation of liquid water by molecular dynamics.
J. Chem. Phys. 60:1545 – 57, 1974.

[Stillinger 75] F. H. Stillinger.
Theory and molecular models for water.
Adv. Chem. Phys. 31:1 – 101, 1975.

[Swan 78] R. J. Swan.
The switching structure and addressing architecture of an extensible multiprocessor, Cm.*
PhD thesis, Carnegie-Mellon University, August, 1978.

[Talukdar 76] S.N. Talukdar.
METAP—A modular and expandable program for simulating power system transients.
IEEE Transactions on Power Apparatus and Systems PAS-95(6):1882 – 91, November, 1976.

[Wallach 74a] Y. Wallach.
Parallel-processor systems in power dispatch—Part I.
Paper No. C74 334 – 9, presented at 1974 Summer Power Meeting.

[Wallach 74b] Y. Wallach.
Parallel-processor systems in power dispatch—Part II.
Paper No. C74 335 – 6, presented at 1974 Summer Power Meeting.

7. StarOS

StarOS [Jones *et al.* 77, Jones *et al.* 78, Jones *et al.* 79] is an *experimental* operating system; it is to be a tool to perform experiments designed to study distributed computation in general, and the exploitation of Cm* in particular. In addition, StarOS itself is to be the subject of experiments exploring message-based and object-oriented operating systems. The experiments of interest are wide-ranging:

Algorithms	What algorithms are suitable for Cm* and similar multiprocessors? How can the availability of many processors be exploited for a performance advantage?
Reliability	Can systems designed for multiprocessors exhibit exceptional reliability? How can the operating system aid in supporting reliable systems?
Resources	What techniques are appropriate for managing distributed resources?
Micro-architecture	What can be done at the lowest levels of machine architecture to enhance the performance and reliability of system and user programs, and also aid in their implementation?
User support	What can be done to aid the user in programming a multiprocessor? How can the system aid the task of programming, managing and interacting with dozens of cooperating processes?
System Structures	What are the performance cost and programming advantages of message-based and object-oriented systems?

The first requirement of an experimental system is that it be adaptable to the changing requirements of the experiments. To this end, the system should be constructed so that modifications are straightforward. The parts of the system that provide a particular function must be readily identifiable so that the function or the means of providing the function can be altered. Also, the system should be extensible so that new functions can be invented and added to the system.

An experimental tool must not occlude the subject of the experiments. An experimental operating system for Cm* must allow the experimenter access to all resources of the multiprocessor. The operating system must not bias experiments by introducing unexpected behavior. Information should not be hidden from the experimenter, but rather the system should aid decision making. The person who designs and builds an experiment should be able to view StarOS as a tool to be exploited, and not as an obstacle to be overcome.

7.1. Features of the StarOS System

R. J. Chansler, Jr.

Four features of StarOS characterize the general nature of the system. The designers believe that a system of this type is particularly suited to the kinds of experiments to be performed.

- StarOS is an **object-oriented** system. Each **object** in the system has a specific type, and the behavior of the object is determined by the functions defined as part of the type. Objects may contain protected pointers to other objects so that objects may be composed into arbitrary structures. The result of a reference to an object does not depend on the physical location of the target, on when or by whom the target was defined or created, or on whether the reference is made by a system or user program.

- StarOS provides a powerful and efficient means for communicating messages among processes. System and user processes share the exact same message facilities. Messages can either be data or pointers to objects.

- There are two classes of functions, each with a specific manner of invocation regardless of the manner of implementation and of what program requests the function. One class of functions comprises the StarOS Nucleus, the basic support supplied by StarOS that is necessary for the execution of any program. Each Nucleus function is **synchronous**: the invoking program must wait until the function has been completed before performing any other action. Each synchronous function is invoked by a particular memory reference. In this regard, functions of the Nucleus are not unlike processor instructions. The other class of functions is the set of **asynchronous** functions: they may be performed concurrently with the continued execution of the invoking program, although a process may choose to wait for the completion of the function. The asynchronous functions are invoked by the transmission of a message. All functions other than the Nucleus functions are asynchronous functions, whether they are part of the StarOS system itself or whether they are provided by user programs.

- **Task forces**, collections of processes that execute cooperatively in order to accomplish a particular task, are created and extended through functions provided by StarOS. StarOS itself is a task force. All functions necessary to create new task forces, or to extend existing ones, are available both to StarOS and to users.

All information in the StarOS system, including programs and data, is contained in **objects**. Each object is of a particular type, and for each type of object, there is a particular set of functions that define how the information represented within the object may be altered or retrieved. For example, the *only* functions that may be performed on a **stack** object are ***Push*** and ***Pop***. Similarly, it is *never* possible to ***Push*** or ***Pop*** a **mailbox** object, it is only possible to ***Send*** and ***Receive*** messages.

Several types of objects are implemented by the StarOS Nucleus. These **representation types** are the basis for building all other structures. Each representation object is implemented as a contiguous segment of memory wholly contained in some computer module. The Nucleus implements all of the type-specific functions, which are synchronous, for these objects. Both users and StarOS can dynamically define new **abstract types**. The physical realization of each abstract

object is some single, specific representation object. Type-specific functions of an abstract type implemented by user or StarOS programs are asynchronous.

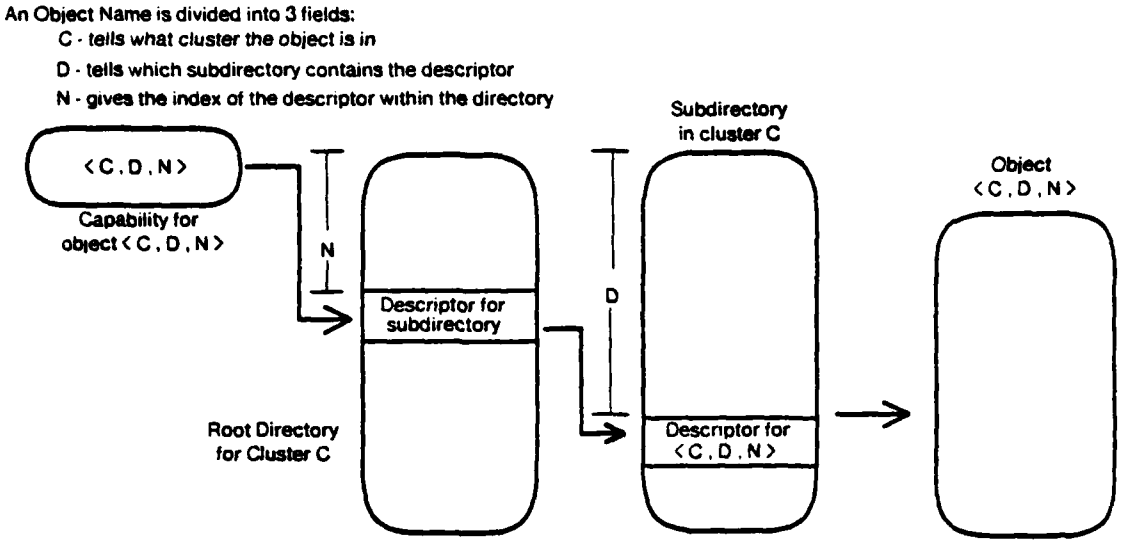


Figure 7-1: How a Capability Points to an Object

7.1.1. Capability Addressing and Uniform Function Invocation

Each object has a unique name which allows the **descriptor** for the object to be located. The descriptor is a record in some **directory**-type object that specifies the representation type, size, abstract type (if any) and physical location of the described object. Each and every reference to an object must specify a **capability** for that object. A capability is a structure that contains the name of an object and specifies (as a bit vector) a list of **rights** for the named object. Each right represents the authority to invoke a type-specific function on the object. Figure 7-1 illustrates that a capability contains the information required to locate a unique descriptor, which in turn contains the physical location of the object named by the capability.

A capability is the only name or address that an object has, and capabilities are the basis of all authority control within StarOS. Consequently, capabilities are protected in that specific functions are provided to manipulate capabilities, and capabilities may never be manipulated as strings of bits. Capabilities are not objects themselves, but are contained within objects just as an integer may be contained within an object. The size of an object is fixed at its creation; consequently, each object has a fixed number of locations, or **slots**, where capabilities may be stored. The slots are collectively

called the **capability portion** of the object; the remainder of the object is called the **data portion.**

One advantage of capability or object addressing is the uniformity and integrity it provides. Quite naturally, whenever a name is required in StarOS, a capability is used. All operating systems and most programs require such object names, and hence must manufacture them for specific purposes if the operating system does not provide them. For example, in Medusa the sender of a message is identified within the message. The form of the name reflects the *physical* location of a data structure. However, if either the data structure moves or the physical location is reused, the name may be that of an entirely different and unrelated process when the receiver eventually sees it. If he replies to this sender, undesirable behavior may result. Fundamentally what is required is a naming facility that is uniform and that has integrity; capability addressing provides such a facility for StarOS.

A **process** is a type of object that may be **assigned** to a processor for execution. The process has access to the objects named by capabilities stored in the process object, and to objects named by capabilities stored in any object to which the process has access.

A process which invokes a Nucleus function may pass a capability as a parameter. This is accomplished not by passing the bit pattern which is contained in the capability, but rather by passing a one- or two-level **capability index.** A one-level capability index designates a capability in the process object; and a two-level capability index specifies a capability in an object which is named by a capability in the process object (see Figure 7-2). The range of capability indexes defines the **capability address space** of the process. Capabilities which are more than two levels removed from the process object may not passed as parameters; however, a process which has the proper rights can copy the capability into the process object. In this manner, objects in an arbitrary graph structure can be accessed.

The sixteen-bit addresses generated by program instructions define the **logical address space.** The programmer views the logical address space of his process as being divided into 16 **windows.** The leading four bits of the sixteen-bit processor-generated address determine which window an address falls into. The remaining 12 bits specify the offset within the window; thus each window is 4K bytes in length. The first fifteen windows may be mapped to objects by associating a capability for an object with a specific window. Addresses in the last window (window 15) are reserved for communicating with the StarOS Nucleus.

The Slocal contains a register for each window. When the process is running, StarOS sets a bit in each register to indicate whether a reference within the corresponding window is to be directed to the module's local memory (a **local** reference) or is to be placed on the Map Bus and communicated to the Kmap (a **mapped** reference). If the register indicates that the window is local, then the register provides a relocation constant that is used to relocate addresses within the local memory of the

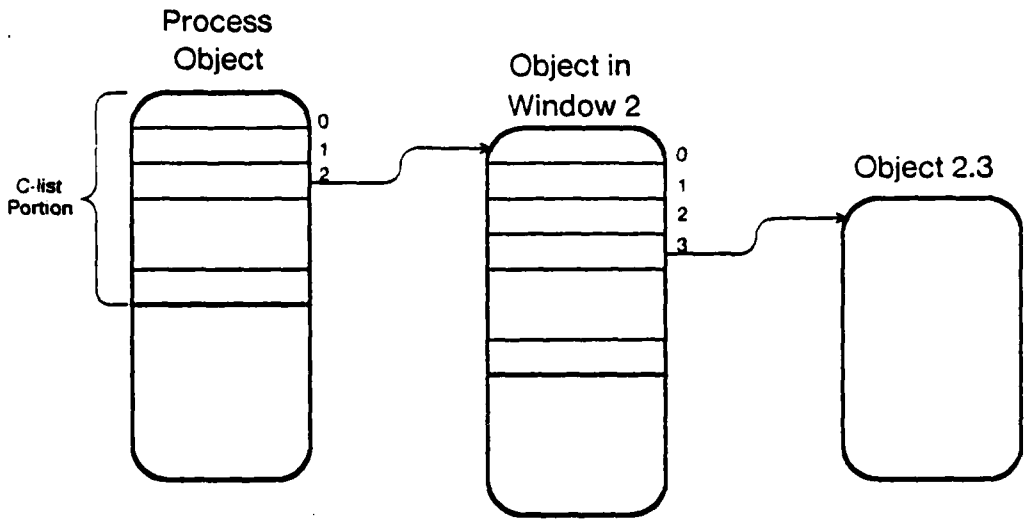


Figure 7-2: Two-Level Capability Indexing

computer module. In this latter case, the window appears as a conventional segment of memory, and "read" and "write" are exactly their usual hardware equivalents. Only the NUCLEUS may access the Slocal registers, and a window is set to local only if it is mapped to a suitable object physically located in the same computer module (more detail is available in Section 7.2.1).

Addresses in window 15 are used to request specific functions from StarOS. In order to invoke a StarOS function, an LSI-11 makes a *write* reference to a location in window 15. All processor-generated addresses are examined by the Slocal, and in the case of an address in window 15, the Slocal does not make a memory reference, but instead notifies the Kmap. The first four bits of the window-15 address are always 1; the Kmap treats the next eight bits as an operation code. The data which the processor has attempted to write to window 15 is used as a parameter. If more than 16 bits of parametersare required for an function, or if the function returns result values, then the data is treated instead as the address of a parameter block, .

For example, the ***Load Window*** function associates an object with a particular window. To invoke it, the process stores the index of a capability for the object at an address in window 15 assigned for that purpose. Any function of the NUCLEUS can be invoked in this fashion.

7.1.2. Structure of the Nucleus

There are two parts to the StarOS Nucleus: the microcode of the Kmaps, and a collection of software processes, one for each processor. The Kmap microcode is about 2200 80-bit records; the code of a nucleus process is about 6K bytes. Together the two parts implement the functions for representation types, and control the assignment of processes to processors. The Nucleus processes also execute the device interrupt routines for the corresponding processor. Nucleus processes are exactly like other processes belonging to either StarOS or a user, except that they are restricted to being assigned only to their corresponding processors, and their assignment can be an immediate effect of a device interrupt, or a processor trap due either to a user request to perform a synchronous function or to a processor-detected error condition.

All Nucleus functions are invoked in a uniform manner; consequently there is no basis for distinguishing between functions implemented in microcode and functions implemented by the Nucleus processes. The Kmap has the *right of first refusal* for all Nucleus functions. If the function is implemented entirely in microcode, then the Kmap performs the function while the processor remains suspended. Otherwise, the Kmap causes the Nucleus process to be assigned to the processor. The Nucleus process completes the function before reassigning the previously executing process to the processor.

StarOS exploits the fact that the Slocal has two sets of registers for mapping the address space of the processor. Whether the processor executes an instruction on behalf of the **kernel space** or the **user space** depends on the setting of a status flag in the Extended Processor Status Word. In this way the address spaces of two processes can be associated with the processor, one with each space. StarOS always associates the Nucleus process with the kernel space. The Nucleus can choose any other process for the user space by specifying a capability for the process and performing a ***Load State*** function.

Some representation types and their functions implemented by the Nucleus are described below:

Basic — Basic objects are composed of an array of words and an array of capabilities. The functions on the array of words include ***Read***, ***Write***, and indivisible ***Increment*** and ***Decrement*** functions. Functions on the capabilities include ***Copy***, ***Transfer***, and ***Restrict Rights***. A basic object is used whenever a conventional segment of memory is needed for programs or data, and is the usual representation for an abstract object.

Deque — A deque object is a double-ended queue of data words. The functions ***Examine***, ***Pop*** and ***Push*** affect one end or the other, depending on the parameter supplied.

Stack — A stack is like a deque except that the functions are restricted to ***Pushing*** and ***Popping*** data from one end. Stacks are used primarily to support the hardware defined trap and interrupt sequences.

Data/CapaMailbox A mailbox object is a queue for messages. A message may be either a single word of data or a single capability; DataMailbox and CapaMailbox are distinct object types. The functions of mailbox objects, ***Send*** and ***Receive***, are described in detail in Section 7.2.3.

Process Each process object has two parts. The **Address Space** is a list of capabilities that the process can reference by index. The **State Vector** is a collection of data and capabilities StarOS uses for the management of the process. Functions are available to ***Activate***, ***Suspend***, ***Block***, and ***Load State*** a process.

Directory Each record in a directory object is a descriptor for some object. There are functions to ***Read*** and ***Write*** descriptors. The descriptors for each directory in a cluster are in the **root** directory for the cluster. The physical location of the root is specified when the cluster is initialized.

7.1.3. Building Task Forces

A task force in StarOS is any cooperating collection ofprocesses, together with their supporting data objects. A task force is typically composed of a set of **modules**, each defining some logical facility. Each process in the task force is created by ***Invoking*** a particular function. For example, except for the Nucleus which implements the special case of synchronous functions, StarOS itself is specified as a collection of modules. The StarOS Task Force is the collection of processes which cooperatively manage the resources of the Cm* machine.

It is possible for a user's task force to include processes created when a user process invokes a function of a module of StarOS. If the user issues a command to ***Stop*** the task force he created, processes created from StarOS modules may or may not be stopped, depending on their relationships with the remainders of the StarOS Task Force and the user's task force.

StarOS and Medusa define use the notion of a "task force" differently. In Medusa, a "task force" has a restricted technical sense: those activities which share a particular descriptor list. The StarOS analog of a Medusa task force is the collection of processes created from the same module object, processes that are closely related in functionality and hence have reason to share information. A StarOS task force may include processes instantiated from different modules, which have quite different functionality. Sharing of information between two processes is determined only by functional need, not by restriction in the memory management that forces any two processes that share one data object to share, in addition, all other data any one of them shares with other processes.

The externally visible attributes of a module are the functions that may be invoked, as described above. For each such function, there is a **function descriptor** that describes how to create a process to perform the function, and an indication of whether a process already exists to perform the

function. To invoke a function, the various required parameters are placed in a basic object referred to as the **carrier**. ***Invoke*** is a function of the type module; it is a synchronous function, and hence is implemented by the Nucleus. The arguments for ***Invoke*** are capabilities for the module and the carrier, and the name of the function. If the module indicates that a new process is to be created when the function is ***Invoked***, the Nucleus implicitly ***Invokes*** the process creation function of StarOS. A capability for the carrier is then sent to the new process as a message. Alternatively, the module may specify that a process already exists to perform the desired function, in which case a capability for the carrier is sent directly to the process as a message. In either case, the difference is functionally undetectable to the invoking program.

The several processes of a typical task force will be derived from the modules that define the problem solution. Whenever a new process is created, it is given a capability for the module specified when the ***Invoke*** function was executed. Thus, many processes may share a module object. The module object may contain the objects used to store intermediate results, or to store the objects used by the several processes for communication with one another.

Modules are the basis for the implementation of abstract types. For each abstract type, there is a module that defines the functions of objects of that type. Such modules are called **type managers.** The functions of a type manager are in one-to-one correspondence with the functions defined upon the particular abstract type. In this way, the solution to a task may be represented as a sequence of functions on abstract objects.

Communication and Synchronization with Messages. Since processes may have capabilities for the same object, it is possible for them to communicate and synchronize using special functions available for the indivisible examination and modification of a shared object. However, since it is *usual* that a task force consists of many processes which must cooperate, StarOS provides special facilities that allow processes to exchange messages and to suspend execution pending the occurrence of particular events. The message system provides mailbox objects—fixed-length buffers for **messages**. Each message is either a single word of data or else a single capability. A particular mailbox can buffer only one type of messages—either data messages or capability messages; which type is determined when the mailbox is created. This permits the programmer to structure a task force so that processes exchange information only along explicit and well-defined channels.

One process may have access to an effectively unlimited number of mailboxes, and each mailbox may have multiple senders and receivers simultaneously holding access to it. Note that by using a capability message, one process can send arbitrary amounts of information, as well as an arbitrarily structured graph of objects. A received capability is sufficient to allow the receiver the ability to access any object in an arbitrary graph structure of objects.

Two different functions are provided for receiving messages from mailboxes. The ***Conditional Receive*** function returns the oldest message, if one is buffered in the mailbox. If the mailbox is empty, the function simply returns. The term "conditional" is meant to imply that the receiving process is interested in receiving a message if one is available, but will go on to perform other actions if no message is available. The ***Registered Receive*** function also returns a message, if one is buffered in the mailbox. However, it acts differently if the mailbox is empty: sufficient information is stored in the mailbox to remember that the caller wishes to receive a message, when one eventually arrives. ***Registered Receive*** then returns a code indicating that no message was received, but that the request was registered.

The ***Send*** function will buffer the message in the mailbox if no receiver has registered interest in receiving the next message sent to the mailbox. If a registered receiver does exist, the oldest registered receiver is dequeued from the mailbox and the message is delivered directly to that process, without being buffered in the mailbox.

Each process has a vector of **events**, each of which may be either *set* or *cleared*. The events are organized into several **event classes**. The ***Block*** function requires as a parameter a mask of the event classes. Unless some event in one of the indicated classes is *set*, then the process will suspend execution until some such event is set. Suspension of execution is never a side-effect of some other function: a process is suspended only if it executes a ***Block*** function.

There is a link between the message system and the event system. The mailbox function ***Registered Receive*** applied to an empty mailbox returns an *empty* result code, but also records in the mailbox that the process desires to receive a message when one becomes available. The process can specify the number of an event to be set if a message is eventually delivered. Any number of ***Registered Receive*** requests may be pending, each associated with a different event. If a process discovers that no further work is possible until a message is received, it can perform a ***Block*** function that waits upon a suitable set of the event classes. In this manner, the message system provides a general structure for synchronizing the processes of a task force.

Delayed delivery of a message to a registered receiver involves delivery of a message to the **portal** of the receiver. A portal is simply a user-designated location, *i.e.*, a word in the address space of the process, or a slot into which a capability message can be stored. Before performing a ***Registered Receive*** on a mailbox, a process will establish a portal by specifying a mailbox, an event name, and a portal name. Event and portal names are small integers between 0 and 255. When a receiver is actually registered in a mailbox, a capability for the receiver process is recorded along with the portal number. **Portal delivery**, the operation of delivering the message, is currently performed by a NUCLEUS process. That is, if the ***Send*** function dequeues a registered receiver to whom the

message being sent is to be delivered, the ***Send*** function stores the message, the receiver name and portal into the address space of the Nucleus process, then forces a trap to Nucleus software that is to perform the portal delivery. Portal delivery simply involves copying the message from the Nucleus space to the portal of the recipient process.

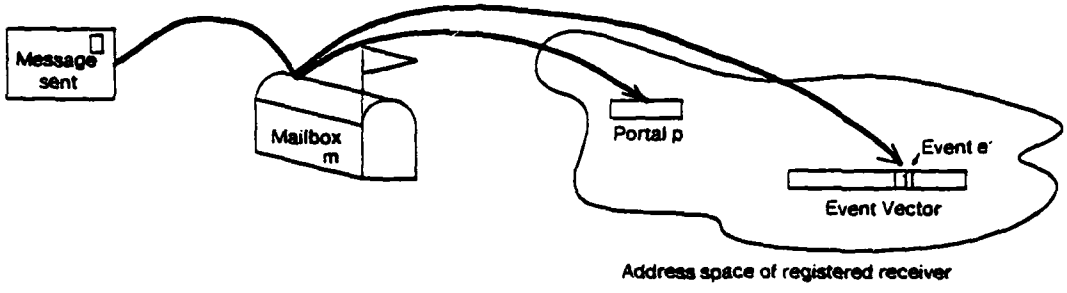


Figure 7-3: Portal Delivery of a Message to a Registered Receiver

To illustrate portal delivery better, consider a process which has associated a portal *p* with an event *e*. It then performs a ***Registered Receive*** on an empty mailbox *m*. Figure 7-3 depicts what happens when, eventually, a message is sent to *m*. It will not be buffered in the mailbox, but will be delivered directly into the portal *p* and the associated event *e* will be set. The process may or may not have ***Blocked*** on an event class that includes *e*. If it did, it may now resume processing.

The StarOS Task Force. StarOS is a task force whose task is the management of the Cm* machine. Aside from the Nucleus, the StarOS Task Force is built from the same components as any user task force. To request service from StarOS, a system or user process ***Invokes*** a function of the particular module that provides the service. StarOS includes modules that create and destroy objects, create new modules from files of program code, maintain a Unix-like file system, operate communications interfaces, create new processes as a result of an **Invoke** request, initialize new task forces from stored descriptions, and supervise the organization of the StarOS Task Force itself.

The typical configuration of the StarOS task forces is such that each cluster is self-sufficient. In principle, a StarOS system could run as an independent system in each cluster, although a particular cluster might be limited by the lack of a particular physical resource—a disk, for example. In practice, StarOS forms a unified system across several clusters. A process that provides some function in one cluster may communicate with its counterpart in each other cluster. In this manner StarOS makes all resources of the system available to every process. For example, in the typical StarOS configuration, allocation of physical memory in a cluster is performed only by A StarOS process within that cluster.

This configuration is not fundamental to the correct functioning of the StarOS system, but was adopted because it facilitated experimenting with robust, fault-tolerant systems. Figure 7-4 illustrates an example configuration within a single cluster.

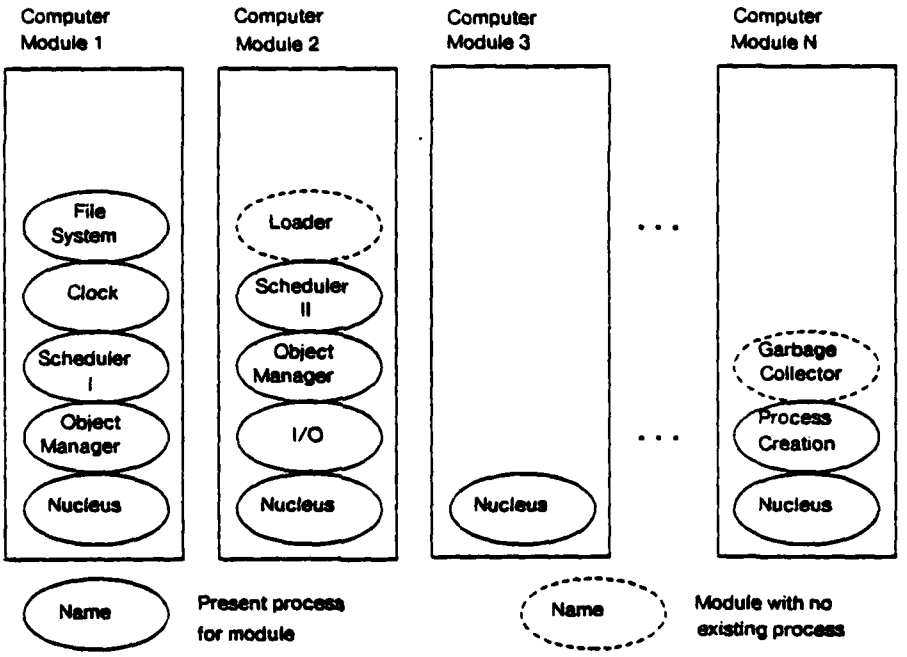


Figure 7-4: Example StarOS Configuration within a Cluster

Within the StarOS Task Force there need not be a process present to serve every function. For example, a process of the Loader module may be created only when there is a request to build a new module. On the other hand, a process that creates new objects must always be present because new objects are required to make new processes. This is in contrast to Medusa where some process ("activity") must be present for each system module ("utility"). In both systems, several processes may share the demand for a particular function.

StarOS processes are not privileged; all authority is granted by specific capabilities. If StarOS is to perform some service for a user process, the user must supply, as parameters, suitable capabilities for the objects to be manipulated. This feature distinguishes StarOS from Medusa in that Medusa grants all system processes the privilege of manipulating any object named in the descriptor lists of a process requesting a system function.

7.1.4. Object Management

The Object Manager is a StarOS module that provides for the allocation of memory, and the maintenance of the the object descriptors within the directory objects. Object Manager functions are performed asynchronously. When an Object Manager function is invoked to create a new object, it allocates storage for the object and returns a capability for the object to the process making the request. All subsequent references to the object are made using this capability or a copy of it. If the object is to be referenced frequently, this capability may be used by a process to associate the object with some window of the process's address space.

In the course of system operation, it may happen that all capabilities for an object are deleted. Such an object is called **garbage**. In this case there is no way at all to reference the object, yet the object still would continue to consume physical memory resources. The Garbage Collector cooperates with the Nucleus and the Object Manager to discover garbage, destroy the objects, and make available the memory resources. Effectively, the Garbage Collector searches all objects for all capabilities, and compiles a list of all objects for which no capabilities exist. The Garbage Collector records the state of its activity by setting the **color** field in the descriptor for each object.

Object management is, for the most part, a cluster-local activity. An Object Manager process allocates objects only within its own cluster; Object Managers in different clusters cooperate by forwarding requests for object creation among themselves. Similarly, Garbage Collector processes in the several clusters can cooperate to perform a systemwide search for garbage. Whenever a capability for an object leaves the cluster in which the physical representation of the object resides, the Nucleus sets the color of that object to *red*. A cluster-local search can be performed to collect non-*red* garbage objects. *Red* objects are immune from garbage collection until a cooperative search is performed by Garbage Collector processes in each cluster.

7.2. StarOS Microcode *Ed Gehringer*

Major portions of the StarOS Nucleus are implemented in Kmap microcode. References by the LSI-11's to non-local memory and other aspects of interprocessor communication must necessarily be performed by the Kmaps, because the only data paths from one LSI-11 to another go through one or more Kmaps. All other functions of the StarOS Nucleus could in principle be performed either by LSI-11 software or Kmap firmware. Where these functions are actually placed depends on certain tradeoffs.

- Firmware runs much faster than software. Precise comparisons are difficult to make because of the dissimilarity of the instruction sets of the LSI-11 and the Kmap, but implementing a function in microcode will often speed it up by a factor of 10 to 20.

- StarOS microcode is written in CMIC assembler code, which means that it is written at a much lower level than the Bliss-11 code which runs on the LSI-11's. Thus coding and debugging for the LSI-11's requires much less of the programmer's time. Mumble, a higher-level language for writing microcode (Section 3.3) now exists, but StarOS could not take advantage of Mumble because it became operational only after substantial StarOS microcode already existed.

- The Cm's have much more memory—64-128K 16-bit words on each LSI-11 vs. 4K 80-bit records of control store in each Kmap. Currently StarOS microcode uses a little more than half of the control store. The designers decided not to exhaust the available control store in order to permit expansion of the microcode for the purposes of system performance measurement, additional Nucleus functions, and support for particular experiments.

- In each cluster there are 10 LSI-11's, but only one Kmap. The Kmaps are sufficiently powerful to perform mapping and synchronization functions without a performance penalty, but as more of the operating system is written in microcode, the Kmaps get busier, and contention for them may develop.

Subject to these constraints, the builders of StarOS have chosen to implement in microcode those functions which provide communication between computer modules or which are critical for system performance. Examples will be described in the following sections.

7.2.1. Mapped References

When a Cm references memory, its Slocal is responsible for determining whether the reference is to be mapped. The Slocal contains two sets of 16 mapping registers, one for each window in user and kernel space. As shown in Figure 2-3 on page 11, each register contains a **map bit**, which determines whether references to the corresponding window are to be mapped. When a ***Load Window*** function is performed, the map bit is turned on so that the first reference to each object will be mapped. When the first reference to a particular object is passed to the Kmap, it determines whether subsequent references to the object will have to be mapped, and, if not, turns the map bit off. Slocal limitations dictate that only a basic object in the local memory that is 4K bytes long can be accessed with unmapped references.[12] The map bit is turned back on the next time that the corresponding window is loaded. The ***Load State*** operation has the effect of performing a ***Load Window*** on each of the windows.

The other bit in each Slocal register is the **read-only bit**. If it is on, then all write references to the window will be passed to the Kmap. For references which are not mapped, the Slocal performs

[12] The first four bits of an address generated by a Cm determine a window number; the remaining 12 bits are used as an offset. Every possible 12-bit offset is a legal address within a 4K-byte basic object. If the object is smaller, the Kmap must be invoked to perform bounds checking.

address translation itself, replacing the first four bits of the processor-generated address with the 6 bits from the Slocal register.

An unmapped reference takes slightly less than 2 microseconds, depending on the type of physical memory. A mapped memory reference to any Cm within the same cluster takes the same 2 microseconds plus an additional 6.6 microseconds, when only a single process is running in the cluster. This is the fastest possible mapped reference, because in this case there is no contention, either for memory, for Kmap microcycles, or for the Kbus. With Kmap saturation but no memory contention,[13] a mapped memory reference took an additional 17.8 μsec. With such Kmap saturation the cost for a mapped reference is 8.6 + m×2, where m is the number of modules generating references. With both Kmap saturation and memory contention,[14] the cost is 50.1 μsec. per memory reference.

Besides the mapped and unmapped references initiated by Cm's, some memory references are initiated by Kmaps in the course of performing operations; for example, reading a capability. Each such memory reference takes about 4.3 μsec.

7.2.2. Capabilities and Tokens

Because StarOS is a capability-based operating system, it must guarantee that a process can manipulate only objects for which it possesses capabilities. This guarantee is enforced by implementing all the support for capability management in microcode, thus assuring that no user can tamper with capabilities in unauthorized ways.

The structure of a capability. Capabilities in StarOS are 32 bits long; that is, two 16-bit words, which are known respectively as the **rights word** and the **data word** (see Figure 7-5). The rights word contains a 3-bit capability type field and up to 13 bits which can be used to specify what rights the holder of the capability has to manipulate the capability and the object it names.

There are four types of capabilities. **Representation capabilities** are used to access representation-type objects. If the proper rights are present in a representation capability, a process with the capability can access or change the state (*i.e*, the contents) of the named object. An **abstract capability** names an object which has been assigned an abstract type. An abstract capability authorizes no access to the state of the object, but it may be used as a parameter to a function of the

---

[13] This was measured when ten Cm's in a single cluster were referencing each other's memory in a ring configuration; that is, Cm i directed all its memory references to the memory of Cm $i + 1$ (mod 10).

[14] In this case, 9 Cm's were all referencing the memory of a tenth. The tenth Cm was not running any code itself.

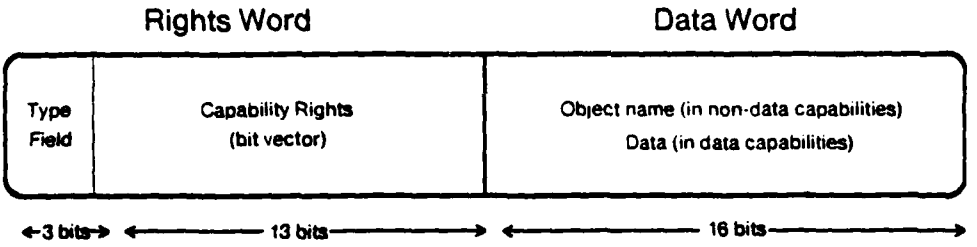


Figure 7-5: The Structure of a Capability

type manager for the abstract type.

There are two types of capabilities that do not name unique objects. **Data capabilities** simply encode sixteen arbitrary bits of information in the form of a capability. **Token capabilities** are merely guaranteed to be created to be unique, that is different from one another. They are used to identify their possessor as having some particular authority. For example, when presented to the NUCLEUS, certain tokens authorize the invocation of special STAROS functions which are unavailable to most processes. Both data and token capabilities may be freely created by any process. Their use is described in greater detail below.

The rights in a capability are represented by a bit vector: if a particular bit is on, it means that the corresponding right is present; and if it is off, the corresponding right is absent. For example, there are 9 rights associated with a representation capability for a Basic object. **Destroy** is required to destroy the object; **Copy** is needed to make a copy of the capability; **Restrict** is required to remove rights from the capability. **Read** and **Write** grant the power to read and write the data portion of the object; **C-list Read** and **C-list Write** apply to the capability portion of the object. **C-list Restrict** is needed to remove rights from the capabilities *in the capability portion* of the Basic object. Finally, any operation which modifies the representation of the object in any way requires **Modify** in addition to the rights which authorize the specific operation. Each time an operation authorized by a particular right is invoked on an object, the Kmap checks to make sure that the capability for that object contains that right; otherwise the operation is aborted.

The data word of a capability usually contains the unique 16-bit **object name** of the object it points at. For some special types of capabilities it serves a different function: for example, in a data capability the 16 bits of data are stored in the data word. The advantage of having it is that it allows one word of data to be stored in the capability portion of an object; otherwise it would be necessary to allocate an object one word long at which the capability could point.

Tokens and Amplification. *Token capabilities are identified by a special bit pattern in their 3-bit capability type field. Tokens do not name objects. Their creation may be freely requested by any process at any time. Typically, a token is used to confirm the authority of its holder to perform a certain Kmap operation which must be carefully controlled for purposes of protection or reliability.*

Section 7.1.3 explained that an abstract object can only be operated on by a small set of functions, collectively known as the type manager. Before accessing the representation of the abstract object, the type manager must first *amplify* the capability for the object. Amplification is performed by the Kmap. The function ***Amplify*** converts an abstract capability into a representation capability for the same object. The function ***Deamplify*** converts a representation capability into an abstract capability for the same object. Both functions require the presentation of the proper *type token* for authority confirmation.

Each time a process creates a new abstract type, it also creates a new **type token**, which is to confer authority to manipulate any object of the abstract type. The process then makes available copies of this token to the functions which it wants to include in the type manager. This token is called a type token because possession of the token identifies a function or process as part of the type manager of the type. Functions which are not part of the type manager do not have a copy of the type token, and thus cannot amplify capabilities of that type.

Other uses of tokens. Authentication of type-manager procedures is an important use of tokens, but there are other cases where a process needs to identify itself. For example, the GARBAGE COLLECTOR needs to be able to read the capability portion of all objects so that it can determine which objects are garbage. For this purpose, it has the GARBAGE COLLECTOR **Token**, which it presents to the Kmap in lieu of a capability for each object whose capability portion it wishes to read. The Kmap itself has a copy of this token, which it compares with the token presented by the GARBAGE COLLECTOR. If the tokens match, the Kmap allows the GARBAGE COLLECTOR to read capability lists. Since no other process has a copy of the garbage-collector token, no other process is able to subvert protection by masquerading as the GARBAGE COLLECTOR.

Similarly, each time the OBJECT MANAGER creates a new object, it needs to be able to manufacture a capability for that object. This means that it must be able to specify the bit-representation of the first capability for the new object. The OBJECT MANAGER possesses the OBJECT MANAGER **Token** which it presents to the Kmap as authority to invoke the function which creates in some capability slot a new capability from a specified bit string.

Capability references. All references to the capability portion of an object are performed by the Kmap. This is guaranteed in the following way: The descriptor of an object contains the base address of the object, the length of the data portion of the object, and the number of capabilities in the capability portion of the object. Using these values, the Kmap can prevent mapped data references from accessing the capability portion of an object. Recall that unmapped data references are allowed only for objects which contain no addressable capabilities at all. Thus, no data references manipulate capabilities, and so capabilities are protected from being overwritten by software.

The **capability operations**, implemented in microcode, are listed below. Measurements are given in microseconds, and also as a multiple of the cost of a typical LSI-11 *Load* instruction (7.4 μsec.) executed without any mapped references. All timings are for in-cluster references, and assume no Kmap or memory contention in the destination Cm.

Create Capability (Representation type) May only be invoked by the OBJECT MANAGER. Manufactures a capability for a newly created object of a representation type.

Create Capability (Data type) May be invoked by any user to create a data capability.

Restrict Capability "Turns off" an arbitrary set of the 13 permission bits in the rights word. This diminishes the power of the capability. This operation may also be used to delete a capability by turning off all 3 bits in its type field.

Amplify Capability Converts an abstract capability into a representation capability. Requires a type token for the type of the abstract object. The most time-consuming part of this operation is comparing the type token with the object type which is stored in the object's descriptor, but is not cached.

Deamplify Capability This is the inverse of ***Amplify***. It replaces a representation capability with an abstract capability. A type token is required, in order to prevent functions other than the type manager from creating abstract capabilities of a particular type.

Copy Capability Copies a capability from one place in the address space to another.

Transfer Capability This is a special form of ***Copy Capability*** which places a new copy of a capability somewhere in the address space, then deletes the old one.

Read Capability Copies information from a capability into the data portion of some object, so that a process may "read the bits", to determine, for example, the type of a capability, or the rights associated with it. This does not subvert protection, because there is still no way for any process to store an arbitrary bit pattern *into* a capability. ***Read Capability*** also furnishes information such as the size of the object, which it gleans from the descriptor. (Often part of the descriptor is cached in the Kmap's data RAM, which will be described in Section 7.2.5. If not, it must be fetched from main memory.)

Load Window Associates an object named by a capability with one of the fifteen windows

of the address space described in Section 7.1.1. An object named by a capability which is in a window is directly addressable: that is, it may be referred to by ordinary LSI-11 read and write references. The Kmap maintains a set of mapping tables, called the **cache**, in its data RAM; and the first reference to an object after it has been loaded into a window will cause the window, and possibly also the descriptor for the object, to be cached. Table 7-1 also notes how long it takes to cache the window, or to cache both the window and descriptor.

Table 7-1: Timing of Capability Operations

Operation	LSI-11 instr. equivalents	μsec.	# refs. to local memory of Cm's
Amplify Capability	18	133.4	13
Copy Capability	13	99.2	10
Create Capability			
Representation	16	120.9	12
Data	12	86.2	9
Deamplify Capability	18	133.4	13
Load Window	9	69.9	8
next mem. ref. (caches window)	+7	+52.0	+5
or, if descr. must be cached too	+13	+94.8	+9
Read Capability	11	78.1	8
if descriptor is not in cache	16	120.9	12
Restrict Capability	9	67.2	7
Transfer Capability	18	130.7	13

Thus, we see that any capability operation or change of addressability can be performed in 23 average LSI-11 instructions or less. It should be noted that this mechanism provides both an expanded address space and support for program modularization, at a relatively small cost. Note that unmapped references have no overhead, and that the second and subsequent mapped references require no indirection through a capability. For example, the distributed PDE application incurs no performance penalty for having the benefit of capability addressing in lieu of the more restricted two-level descriptor-based addressing offered by SMAP and MEDUSA.

The cost of object addressing is likely to be significant only for processes whose working set exceeds the capacity of the window registers, resulting in frequent invocations of ***Load Window***.

Such processes would expect to incur substantial overheads from any mechanism which attempts to relieve the constraints of the LSI-11's small address space.

It is worthwhile to scrutinize an individual operation to assess how resources are expended in performing the operation—in particular, to account for the seemingly large number of references to the memory of Cm's during the operations listed in Table 7-1. First, we observe that execution time for a StarOS micro-routine is divided approximately equally between microcycles and memory references.

Now, we consider the ***Amplify Capability*** operation, which performs 13 memory references. It takes two parameters: The first is a capability index (see Section 7.1.1) which tells which capability to amplify; and the second is a pointer to the type token which matches the type of the object named by the capability. These two parameters are contained in a parameter block (see page 115). The first memory reference by ***Amplify Capability*** reads the *processor data*, the address of the parameter block (memory reference no. 1). Then the capability index (2) is read from the parameter block. Using this index, the capability itself (3, 4) is read. A special value called a **plug** is written into the capability's type field (5) to flag the fact that this capability is in the process of being modified, so that no other Kmap operation attempts to modify it in the meantime.

The abstract type (6) of the object named by the capability is read from the descriptor for the object because abstract-type names are not kept in the cache. The second parameter (7) names a type token, which is then read (8, 9). The amplified capability (10, 11) is written back, thus overwriting the plug that says that the capability is being operated on. Then a zero is written into the first word of the parameter block as an indication that the operation completed successfully (12), and the Cm is awakened (13).

Most other capability operations follow a similar sequence of actions, though they are somewhat cheaper because there is no need to read a token and—if the capability is not to be rewritten or deleted—to write a plug. Since the microcode performs some of the functions of a conventional executive or supervisor program, it is not surprising that a microcode function has corresponding entry and exit overheads to read parameters and write a result code. This overhead is typically 4 or 5 mapped memory references (about 35 – 45 μsec. or 5 – 6 locally executed LSI-11 *Load* instruction equivalents). The corresponding overhead for a function implemented by the Nucleus process is about 750 μsec. or approximately 100 instructions.

7.2.3. Message Transmission Facility Measurements *Anita Jones*

As mentioned earlier, StarOS and the applications that run on the base it provides are both routinely composed of distributed processes that communicate via messages. Transfer of parameters during function invocation is accomplished by message communication. Because StarOS is object oriented, a StarOS process executes with an address space populated only with those objects that it needs to perform its function. This is a desirable attribute, say, from the perspective of building reliable software, but it requires more transmission of objects between address spaces than would be necessary in a conventional system where related processes share large amounts of data. Such transmission of objects to tailor address spaces is usually accomplished by sending capabilities for the objects as messages. For all these reasons the usability and the performance of the message transmission facility is crucial to determining overall system behavior. In this section we describe in more detail the message communication facility of the system and initial results from measurement of its performance.

Table 7-2 presents measurements of the three message functions ***Send***, ***Conditional Receive*** and ***Registered Receive***, which were described in Section 7.1.3. Separate entries have been made for cases in which the functions perform significantly different work. Basically, the cost differences between capability and data message functions are due to the fact that each time a capability is read or written, two memory references are required; data messages require only a single reference.

Table 7-2: Timing of Message Primitives

INST = LSI-11 instruction equivalents (1 = 7.4 μsec)
μSEC = Microseconds
REFS = Number of memory references

	Data Messages			Capability Messages		
	INST	**μSEC**	**REFS**	**INST**	**μSEC**	**REFS**
Conditional Receive						
message is returned	16	118	11	24	180	15
no message available	15	113	10	24	181	16
Send						
message is buffered	15	110	10	20	151	14
registered receiver found	27	197	19	33	245	23
Registered Receive						
message returned	21	157	12	25	186	15
record receiver in mailbox	22	166	16	32	235	22

The most important aspect of the measurement is that the cost of the functions is some modest multiple of the cost of the fastest possible *Load* instruction on Cm*. Typically, the cost of performing such a function is measured in hundreds of *Load* operations, not because the message transmission function is complex, but because in most systems the cost is driven up an order of magnitude by the cost of entry and exit to the nucleus or kernel of the operating system. For a message-based system to perform adequately, message transmission must be inexpensive.

All of these measurements include the cost of reading the parameters required to perform the functions. Each function requires the specification of a mailbox and some value (16 bits) to indicate where to fetch a message from or how to effect delivery. ***Registered Receive*** has three parameters: the mailbox name, the address of the portal site in case a message can be delivered there immediately, and a small portal number that will be used if the receiver is registered. (The latter is sufficient to determine both the portal site and the event associated with the portal when porta delivery actually occurs.)

Measurements also include writing one word of status back to the process to communicate th action taken during execution of the function. These entry and exit actions account for between 12 and 30 per cent of measured function execution time.

*Measurements of **Portal Delivery** are not included here.* We chose not to include it as part of the *initially* microcoded portion of the Nucleus. After gaining some experience with the message system, we may choose to convert the software implementation of portal delivery to microcode. Future measurements will include frequency of usage of the message functions, cost of ***Portal Delivery***, and a survey of just what sorts of messages are being sent. We would like to determine that relative usage of data to capability messages, for example.

7.2.4. Additional Facilities Provided by the Kmap

Synchronization support. It is the Kmap that provides synchronization for Cm* systems. In particular, if an object is mapped, the Kmap in the cluster in which the object's physical representation exists will assist in performing operations on that object. A cached descriptor contains a lock. The Kmap uses that lock to order the actions performed by simultaneously executing contexts. The Kmap routines utilize these locks to avoid undesired interleaving; they permit a Kmap can make multiple references to an object in an indivisible way. For example the several words of a descriptor or a capability are written indivisibly.

The directory entry for each object (Section 7.1.4) contains a **lock**, which is used by the StarOS microcode any time a function needs exclusive access to an object. In the optimal case, where the

descriptor to be locked is cached, and there is no contention for the Kmap, 25 microcycles, or 3.9 μsec., is required to locate and lock a descriptor. The Kmap "remembers" the location of the descriptor so that it can unlock it after it is finished with it; unlocking takes only 2 microcycles. No function requires that more than one lock be set simultaneously, nor is the processor requesting the function permitted to proceed while the lock is held.

One of the many uses of locks is support for the indivisible ***Increment*** and ***Decrement*** operations used in process synchronization. These operations take two parameters, a capability for the object containing the word to be decremented, and the offset of the word within the object. The ***Increment*** operation adds one to a word of memory, assuring that only one process at a time can be performing an ***Increment*** or a ***Decrement***. The ***Decrement*** operation similarly subtracts one from a word of memory, unless the word was already zero. In any case, the old value is returned to the requesting process, which may decide to block if the value was zero. The ***Increment*** operation takes 89.9 μsec. and the ***Decrement*** operation takes 101.2 μsec., not including the Cm time used to set up the parameter block.

Operations on additional representation types. Table 7-3 gives sample timings for function of other representation types: stacks, deques, and directories.

Table 7-3: Timing of Operations on Representation Objects

Object type/ operation	LSI-11 instr. equivalents	μsec.	# refs. to local memory of Cm's
Stack			
Push	4	32.6	4
Pop	5	39.8	5
Deque			
Push onto front	6	46.4	6
Push onto rear	6	47.1	6
Pop from front	5	39.3	5
Pop from rear	5	40.1	5
Directory			
Read	2	15.5	2
Write	3	22.1	3

Garbage-collection support . The Kmap is responsible for maintaining the special data structures used by garbage collection. For example, each time a capability for an object is created or copied while the GARBAGE COLLECTOR is running, the object is placed in a special queue, so that the GARBAGE COLLECTOR will not mistake it for garbage. This requires an additional 35.7 μsec. Whenever a capability is copied into another cluster, the object it names is marked "red", which takes 20.1 μsec. Further, as explained in the discussion of tokens, the microcode contains functions to allow the GARBAGE COLLECTOR to read capabilities without the need for individual object capabilities.

Miscellaneous operations. The microcode also contains a special routine for bypassing the capability addressing structure and writing to absolute memory addresses anywhere in the system. This code is used for initialization and reconfiguration, and may not be invoked except by a process presenting a special token. The ***Load State*** operation, mentioned in Section 7.1.2, is also implemented in microcode. It ensures that all the map bits in the Slocal registers are on, and invalidates all of the cached capabilities in the windows. This is necessary to prevent the new process from gaining access to any objects which belonged to the old one. This operation does not require a special token, but it is not allowed to be invoked from user space.

7.2.5. Implementation Considerations *Steve Vegdahl*

StarOS use of the Kmap Data RAM. The Kmap data RAM contains 1024 eighty-bit records of bipolar memory which can be accessed in less than two microcycles without interrupting the Kmap. Each record is divided into five 16-bit words. Half of the data RAM is used for caching descriptors, so that frequently accessed objects can be referenced without reading the descriptor from main memory. The savings are substantial: when the descriptor is present in the cache, it can be located in about 19 microcycles (depending on how many cache locations have to be searched) and no main-memory references, but if it is necessary to read it in from main memory, typically about 187 microcycles and 4 memory references will be needed. This represents a saving of about 46.8 μsec. per object reference.

Twenty percent of the data RAM holds information about capabilities which have been loaded into windows. This information includes the object name of the capability in the window, along with read, modify, and write rights. Each window entry contains a link to the cached descriptor it references so that no cache-searching is required when objects in windows are referenced. The data RAM also contains other miscellaneous information such as the name of each process currently loaded onto a computer module in the cluster and the kernel Kmap registers.

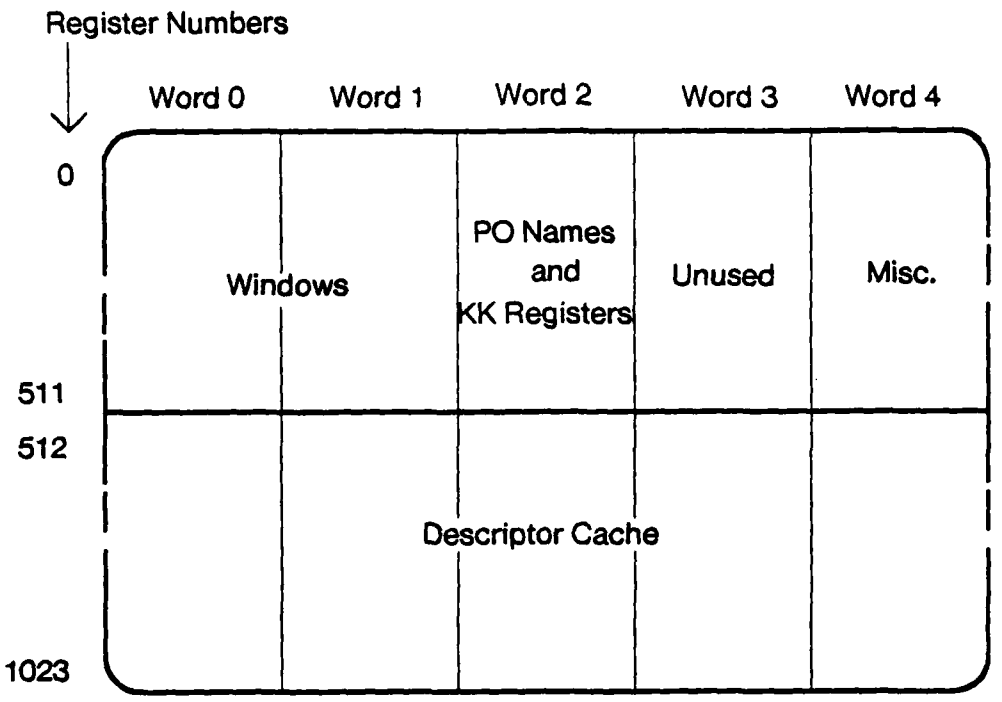


Figure 7-6: The Data RAM as Used by StarOS

Deadlock and starvation avoidance. There are a number of deadlock and starvation issues which must be addressed by the microcode (see Section 4.4). The major possibilities arise from interactions among instances of one or more of the following resource allocations:

1. A computer module acquiring a Kmap context.
2. A context locking a descriptor.
3. A context acquiring an Slocal for the purpose of making a memory reference.
4. A context acquiring a second context in foreign cluster for the purpose of having some work done in that cluster.

In each of these scenarios, either a processor or a context has the potential for starvation. In addition, there may also be deadlock problems when more than one processor or context is involved. The following set of microcode conventions is used in order to prevent starvation and deadlock:

- Currently, computer modules busy-wait on Kmap contexts, making sure that context 7 is never used to wait for a cross-cluster reference to complete. Thus, it can never happen

that all eight contexts in one Kmap are waiting for the completion of cross-cluster references from a second Kmap, at the same time that all eight contexts in the second Kmap are awaiting the completion of cross-cluster references from the first Kmap. This solves the deadlock problem, but not the starvation problem. Eventually, the starvation problem will be solved by priority ordering of requests. For now, we count on the fact that all computer modules run with time-outs disabled.

- Slocal allocation is always done at a lower level than descriptor locking. In other words a context will never allocate an Slocal, and then try to lock a descriptor without having first released the Slocal.

- Intercluster context allocation is always done at a higher level than descriptor locking. This means that a descriptor may only be locked by a context in the cluster where the descriptor resides, and that a context may not lock a descriptor and then wait on a cross-cluster event.

- A context never locks more than one descriptor or allocates more than one Slocal or intercluster context at any time.

- The hardware guarantees that competition for contexts among local computer modules and intercluster requests will be starvation free.

Microcode design. The philosophy behind the design of the StarOS microcode was to write the code in a modular way, making heavy use of procedures (and macros, to the extent that the microassembler allowed) to manipulate data structures, rather than spreading the information throughout the source code. In addition, we attempted to make the data structures in the data RAM simple in order to minimize the interaction among modules which use closely related data structures. (For example, in the original version of the StarOS microcode [Jones *et al.* 77] there were eight types of links in the data RAM. There is only one type of link in the current version.) Intercluster communication is also done through a small number of intercluster primitives. This was motivated primarily by a desire to make the code easier to maintain and debug; bottlenecks could be optimized out once an operational version of the microcode had been completed.

Two aspects of this strategy are significant. First, the resulting code is conservative of space in the instruction store of the Kmap, using only 2200 of the 4096 available records (54%). This allows the option of adding functions to the Nucleus, moving functions from the Nucleus processes to the microcode, introducing code for performance measurement, or adding special microcode to support individual experiments. Second, the resulting speed of the microcode is not optimal. However, since the system became operational, careful performance measurements have been possible. With specific knowledge of the inefficiencies that are important *in practice*, effort could be directed at improving performance while retaining the advantages of modularity.

Four specific optimizations have been made. In each case, the optimization increased the number

of instructions, and thus would have been impossible had the instruction store been exhausted.

- *Message operations.* The first version of the message operations called to the pointer routines which did pointer manipulation for deques and stacks. The pointer routines performed the proper function, but did so inefficiently for the purpose of the message operations. For example, each pointer routine read the pointers from main memory even if a previously called routine had just read them. Two additional routines were written which manipulated the pointers in an efficient way for the message operations. Resulting improvement was roughly 15 to 20 per cent.

- *Cross-cluster memory references.* The initial version of the microcode optimized only intracluster memory references. Intercluster references originally cost 37 μsec. One reason is that the intercluster-reference code invoked the single general-purpose routine which packed and sent all linc messages. That generality was too expensive. In order to optimize the simple cross-cluster memory reference, a specially tailored linc-message sequence had to be rewritten. At this writing, the intercluster reference costs 30 μsec. A highly optimized intercluster reference has been written but not debugged. Its estimated cost is 23 μsec.; that is, it will execute at essentially the same speed as the cross-cluster references in the Smap microcode.

- *Slocal arbitration.* There must be provision in the microcode for arbitration among contexts requiring access to the same Slocal (same computer module) for a memory reference. The initial version of the arbitration used the same starvation-free spin-lock routines that were used to arbitrate descriptor-lock contention. Experiments showed that this hampered performance greatly when contention was high. Consequently, a new arbitration scheme was implemented; it is discussed in Section 10.1.3.

All the above optimizations have been made in a modest amount of time. Because the microcode was originally written in a modular form, replacement of code sequences has been simplified. Such optimizations can be debugged in the context of an otherwise debugged system. There is one more benefit to be noted. We are looking forward to performing a wide variety of experiments. Because the microcode is so modular, usually only a single routine is responsible for performing some action or detecting a particular state. Hence, performance probes are expected to be relatively simple to install.

7.3. Future Work

R. J. Chansler, Jr.

Future work for the StarOS project will be directed along two fronts. Additional general purpose user utilities need to be constructed to make StarOS a more comfortable environment for programmers. This work will include the development of a flexible and systemwide I/O system, a generalized loader for creating the initial processes and objects of a task force, an interface to the local Ethernet, and a StarOS-specific version of the Six12 debugger that understands how multiple processes are created from a module.

StarOS is an experimental operating system. It is anticipated that this aspect of the system will be exploited by members of the StarOS group in exploring their own research interests, and by other members of the community investigating application systems for multiprocessors. Some of the experiments that are planned by members of the StarOS group include the following:

- Collection of statistics concerning the use of objects and capabilities will provide useful information for the design and optimization of object-oriented systems. This information will be of particular value for comparison with similar data from another object-oriented system, C.mmp/Hydra [Almes 80].

- Studies of the design of the Kmap microcode will serve two purposes. Using the data presented above and operation frequency measurements, it will be possible to learn how to better optimize the performance of the Kmap as an integral part of the StarOS system. Second, these investigations will useful information for the designers of future microcoded processors.

- An important motive in the development of multiprocessors is the potential for improved reliability. Work in progress is directed towards learning how to improve the reliability of the StarOS system itself, and how to design and build user tasks forces that exploit the features of a multiprocessor system for enhanced reliability.

- Many tasks that seem suitable for multiprocessor solution involve the transfer of large amounts of data from secondary stores. Research is directed at discovering methods for the efficient distribution of data, and methods for the convenient specification of how data is to be distributed to the processes of a task force.

- In order reduce the difficulty of specifying the configuration of task forces, the Task specification language is being developed ([Jones and Schwans 79]). Part of that work will include measurements of how well automatic initial placement and assignment heuristics accomplish their task.

- StarOS itself provides an example of a complex distributed system. Studies are planned which explore the dynamics of such systems, particularly with regard to the management of system resources.

Part of the success of StarOS depends upon whether the system provides an attractive environment for people to explore multiprocessor solutions to a variety of tasks. Consequently, it is the intention of the StarOS group to aid and encourage other investigators in using the Cm*/StarOS facilities. Preliminary studies for several tasks have been undertaken.

7.4. References

[Almes 80] G. T. Almes.
Garbage collection in an object-oriented system.
PhD thesis, Carnegie-Mellon University, June, 1980.

[Jones *et al.* 77] A. K. Jones, R. J. Chansler, Jr., I. Durham, P. Feiler, and K. Schwans.
Software management of Cm*—a distributed multiprocessor.
In *National Computer Conference, Proceedings, Vol. 46*, pages 657 – 63. AFIPS, 1977.

[Jones *et al.* 78] A. K. Jones, R. J. Chansler, Jr., I. Durham, P. Feiler, D. A. Scelza, K. Schwans, and S. R. Vegdahl.
Programming issues raised by a multiprocessor.
Proceedings of the IEEE 66(2):229 – 37, February, 1978.

[Jones *et al.* 79] A. K. Jones, Robert J. Chansler, Jr., I. Durham, K. Schwans, and S. R. Vegdahl.
StarOS, a multiprocessor operating system for the support of task forces.
In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 117 – 27. ACM/SIGOPS, Pacific Grove, California, December 10 – 12, 1979.

[Jones and Schwans 79]
A. K. Jones and K. Schwans.
TASK forces: distributed software for solving problems of substantial size.
In *4th International Conference on Software Engineering*, pages 315 – 30. ACM/SIGSOFT, Munich, Germany, September, 1979.

8. MEDUSA

Pradeep Sindhu

MEDUSA [Ousterhout *et al.* 80] is a multi-user operating system developed for Cm*. The project is an attempt to understand the effect of the distributed Cm* hardware on operating system structure, and to build a system that capitalizes on and reflects the underlying hardware. MEDUSA is the second operating system written for Cm*. The first system, STAROS, has been described in Chapter 7 of this report. The two operating systems have been motivated by different concerns and have approached their designs from almost opposite directions. The design of MEDUSA was driven by the desire to exploit the modularity and robustness of the Cm* hardware, both for the purpose of providing these attributes within the operating system as well for making them available to user programs. STAROS, on the other hand, took a top-down approach in which facilities provided to users were much more important in determining the design of the operating system than features of the underlying hardware.

The goal of the MEDUSA project has been to produce an operating system with three specific attributes: *modularity*, *robustness* and *performance*. Although these attributes are general enough to be goals for most systems, what makes them interesting in the context of MEDUSA is the way in which they interact with the hardware of Cm* to influence the structure of the operating system.

Instead of starting out with an *a priori* notion of the system structure required to achieve the above goals, we allowed the underlying architecture to determine the structure. Two aspects of the Cm* hardware have been especially important to the design of MEDUSA. First, the components of Cm* are physically distributed such that the cost of accessing different portions of the system from a given point is not uniform. Second, the computing elements are connected together by powerful communication controllers that can be used to implement a wide variety of communication and addressing mechanisms. This combination of *distribution* and *sharing* present in the hardware gives rise to corresponding issues of *partitioning* and *communication* in the structure of MEDUSA: how should the operating system be partitioned to enhance its modularity and to allow its components to make efficient use of the distributed hardware? How should the components communicate so as to function in a robust way as a single logical entity? An examination of these issues led to an operating system structure that has the following properties:

- Functions of the operating system are partitioned among disjoint **utilities** that are distributed around the hardware; each utility implements some abstraction for the rest of the system.
- Utilities are organized as **task forces**, which are sets of closely cooperating parallel **activities**; user programs are also expected to be organized in this way.
- Utilities use **messages** to communicate with each other and with user programs.

It is interesting that the approach followed in the design of MEDUSA has led to an operating system structure that resembles the structure of the hardware quite closely. Task forces in the operating system are analogous to clusters of Cm's in the hardware, and activities in a task force are analogous to processors in a cluster. However, task forces are not constrained to map to clusters in any way—a single task force may have its activities spread around on several clusters. The parallel between the hardware and operating system also extends to the internal structure of the Kmaps and utilities. Each utility activity is internally multiplexed in much the same way as the hardware multiplexing of contexts in the Kmap.

The implementation of MEDUSA consists of three distinct components. *Microcode* running on the Kmaps provides the addressing and communication mechanism. A small *kernel* that resides in every processor provides for activity multiplexing and interrupt handling. The *utilities* provide the functions of file management, memory management, task force management, exception reporting and debugging. A significant portion of the remainder of this chapter will be devoted to the design of the microcode, and to preliminary measurements of its size and performance. The functions provided by the MEDUSA kernel will also be described, but only briefly. The overall design of the utilities and the functions implemented by each utility will be taken up in the last part of the chapter and preliminary measurements of their size and performance will be presented.

8.1. The MEDUSA Microcode

The presence of powerful microprogrammable Kmaps in the Cm* architecture has had a strong influence on the structure of the MEDUSA operating system. MEDUSA is partitioned along functional lines into a set of utilities that use messages to communicate with each other and with user programs. Although this structure was motivated by physical distribution in the hardware of Cm*, it was feasible primarily because the power and flexibility of the Kmaps permitted the efficient implementation of a general message mechanism as well as of shared memory.

In deciding whether a function ought to be implemented in Kmap microcode, the most important consideration was, of course, whether the performance of that function significantly affected the performance of the operating system as a whole. Message communication and address mapping are two functions that are critical to the performance of MEDUSA and are therefore implemented entirely in Kmap microcode. The message mechanism is crucial to achieving not only performance, but modularity and robustness in MEDUSA as well. The strong separation between utilities that is achieved by making use of messages for inter-utility communication is largely responsible for the modularity of the system. Robustness is attainable, at least in principle, because it is not possible for utilities to contaminate each other in arbitrary ways since they do not share read/write memory.

The message mechanism alone, however, is not sufficient to support a structure like that of MEDUSA. Activities within the same task force need to interact on a much finer grain than is permitted by the use of messages, and such fine-grain communication can usually be supported only by a simple mechanism such as shared memory.

Generally speaking, all of the facilities that the microcode provides for the operating system are available to user programs as well, although users may not invoke microcode functions that require *utility privilege*. Furthermore, microcode functions are callable directly by user programs, so there is no loss of performance compared to calls made to the microcode by utilities.

The design of the MEDUSA microcode has had to acknowledge the distributed nature of the hardware that it executes on. In order to exploit the distribution present in the hardware and to avoid performance bottlenecks and poor reliability, the microcode treats all Kmaps on an equal basis. It imposes no *static* master-slave relationships between Kmaps, although it permits master-slave relationships to exist temporarily during the life of particular operations. The Kmaps in the system cooperate closely and function together to provide the operations implemented by the microcode The design of the microcode in this distributed environment has led to a number of interesting problems that are relevant in a broader context:

- How should the message communication mechanism be implemented so it can function efficiently and reliably?
- What steps should be taken to eliminate starvation and deadlock over shared resources?
- How should the object name space be kept consistent, given that names of objects are distributed, and that there may be multiple names for the same object?
- How should exception handling be organized so that it is (a) effective and (b) implementable within the microcode?

The techniques used to solve these problems in MEDUSA will be discussed in some of the later sections on the microcode. First, however, the address structure of MEDUSA will be described in Section 8.1.1. Section 8.1.2 then goes into the design of the message mechanism and pinpoints the features that make it efficient. The problems that arose because of the distributed nature of Cm* and the solutions used in MEDUSA for these problems are presented in Section 8.1.3. The design of exception handling in the microcode is dealt with in Section 8.1.4. Finally, Section 8.1.6 presents statistics on the performance and size of the MEDUSA microcode.

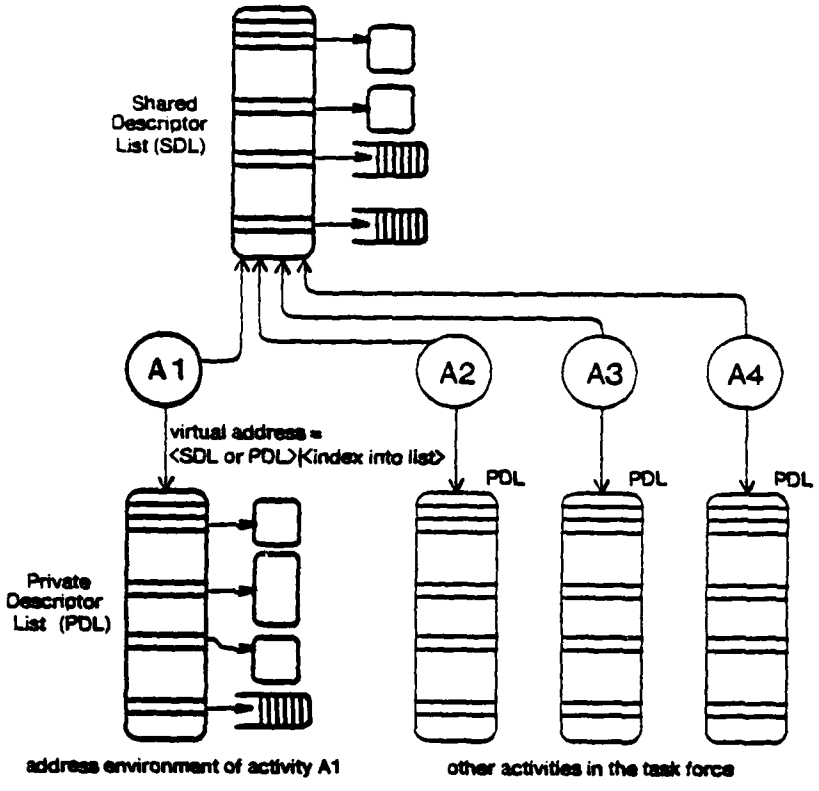


Figure 8-1: The Address Structure of a Task Force

8.1.1. The MEDUSA Address Structure

Since the individual processors of Cm* are slow, most interesting programs that run on Cm* will need to use several processors. MEDUSA provides an explicit structure, called the task force, to support this parallelism. A task force is a collection of activities that cooperate closely to perform a given computation. An activity is the entity that gets scheduled on a processor; it is the MEDUSA analog of what is called a process in most other operating systems. The main difference between the notion of an activity and a process is that an activity may not exist independently of a task force; furthermore, there is an implicit assumption that the activities inside a task force need to interact frequently.

Information manipulated by MEDUSA activities is stored in objects that are addressed using

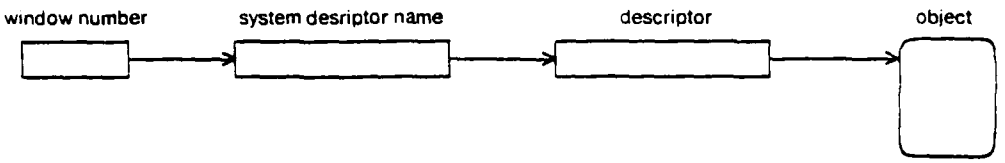


Figure 8-2: The Logical Mapping of Addresses in MEDUSA

descriptors. Descriptors contain the type, location, and size of objects and are kept in protected objects called **descriptor lists.** The descriptors for objects accessible to the activities in a task force are organized into a number of descriptor lists as shown in Figure 8-1. The virtual address space of each activity is defined by the contents of two descriptor lists: the **private descriptor list** (PDL), to which the activity has exclusive access; and the **shared descriptor list** (SDL), to which all activities in the task force have access. Thus virtual addresses generated by an activity consist of a *list name* (PDL/SDL), an *index* into the list specifying a descriptor for the object being referenced, and an *offset* into the object.

The processors in Cm*, however, are capable of generating only 16-bit addresses. In order for the system to map a 16-bit processor-generated address into the full physical address[15] of the object being referenced, there must be a way of associating the processor's address with the corresponding virtual address in the space of the activity making the reference. The microcode provides an operation called ***Load Window*** that allows an activity to bind a descriptor in its virtual address space to one of sixteen **windows** in the processor's address space.

Once this binding is established, hardware in the Slocal cooperates with the Kmap to map a processor-generated address into the corresponding physical address. Local references are sent out onto the LSI-11 bus of the processor making the reference, whereas non-local references are shipped out to the Kmap. Logically speaking, the Kmap converts the window number specified by the processor's address into the address of a descriptor which it uses to fetch the descriptor from the appropriate descriptor list. It then uses the physical address in the descriptor to access the object pointed to by the descriptor (Figure 8-2). In practice, however, the reading of the descriptor from a descriptor list is bypassed most of the time by maintaining a cache of frequently used descriptors in the Kmap (Figure 8-3).

---

[15] A complete physical address in MEDUSA is 26 bits long: 4 bits of *cluster number*, 4 bits of *Cm number* within the cluster, and 18 bits of *memory address* within the Cm.

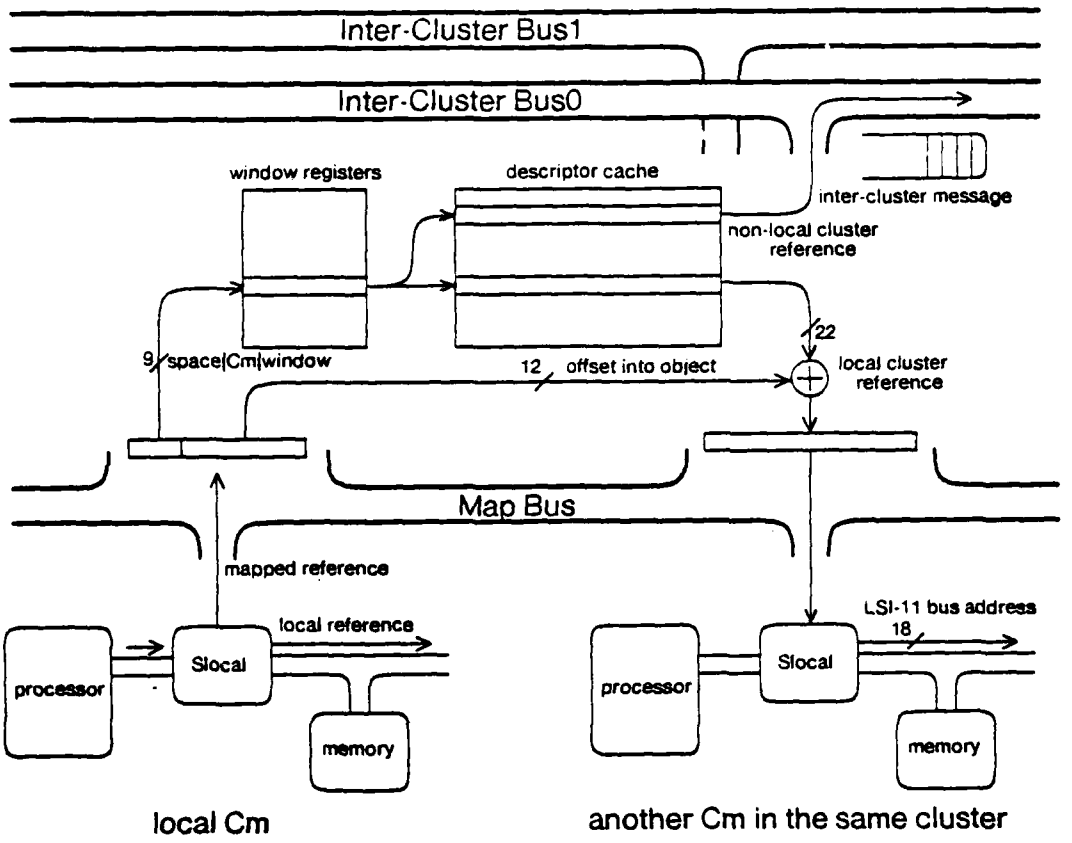


Figure 8-3: The MEDUSA Mechanism for Non-Local References

Two aspects of the design of MEDUSA's address structure deserve to be elaborated further. First, information that is used by the system to map virtual addresses into physical addresses is not maintained in one physically central place. Rather, it is *distributed physically* on the basis of ownership. Thus descriptors for objects private to an activity are kept in a private descriptor list that is present *only* in that activity's virtual address space; descriptors for objects that are shared by the activities in a task force are kept in a shared descriptor list that is present *only* in the virtual address space of the activities belonging to the task force. The motivation for distributing this information physically is to avoid the performance penalty and poor reliability that would result from keeping one or at most a few structures that are crucial to the mapping process. Since mapping information is stored with its owner rather than within the system, failures in the portion of the system that manipulates descriptors is less likely to damage unrelated descriptors. As a general rule, information in MEDUSA that is not related logically is not kept together physically either—freedom from such

arbitrary system-imposed associations is considered fundamental to achieving robustness within the operating system.

The address structure also provides a simple protection mechanism that allows the above principle to be applied to the storage of objects that are implemented by the MEDUSA utilities. This mechanism permits such an object to be stored with the owner of the object, yet to be manipulated by the utility that understands its internal structure. In addition to reducing the likelihood of an errant utility destroying an object it manipulates, the mechanism ensures that the size of a utility does not grow with the number of objects it manages.

A second attribute of the MEDUSA address structure is its relatively flat nature. MEDUSA has one less level of indirection from virtual addresses to physical addresses than does STAROS: MEDUSA's mapping proceeds from descriptor specifier to descriptor to object whereas STAROS's mapping proceeds from capability specifier to capability to descriptor to object. If an object is shared, MEDUSA has one descriptor for each co-owner whereas STAROS has multiple capabilities that point to a single descriptor for the object. The primary virtue of a flatter address structure is that it is simple to implement in microcode. A capability-descriptor structure like that of STAROS is more difficult to implement because two distinct name spaces, the capability space and the descriptor space, need to be managed instead of just the descriptor space. However, for its trouble, STAROS gains the ability to share objects in a much more controlled fashion than is possible in MEDUSA.

Another disadvantage of the STAROS's address structure is that descriptors are kept in a few, centralized descriptor lists. Since descriptors used by unrelated processes are stored together physically, a failure in one of these descriptor lists (or in the code that manipulates them) could affect completely unrelated processes, making recovery from such errors quite difficult. MEDUSA avoids this problem since descriptors are stored in the same descriptor list only if they belong to the same activity or to the same task force.

A consequence of having more than one descriptor for an object in MEDUSA is that a shared object is difficult to move since information about its location is distributed. The solution used is to keep pointers from an object back to the descriptors for the object so that all of the descriptors can be found and changed when the object is to be moved.[16] In contrast to a capability-descriptor name space, the presence of backpointers does not affect the complexity of the microcode since the microcode has no knowledge of their existence—they are implemented entirely by the memory manager utility of MEDUSA.

---

[16] These backpointers also turn out to be convenient for handling exceptions on a shared object since they explicitly identify the activities that can access the object. Thus when an exception occurs on a shared object the activities sharing the object can be easily notified.

8.1.2. The Communication Mechanism

The communication mechanism implemented by the microcode is fundamental to achieving the performance, modularity and robustness goals of MEDUSA. This mechanism incorporates a number of features in its design that make it efficient and powerful enough to be used widely within the operating system as well as in user programs. In contrast with most operating systems MEDUSA's message mechanism acknowledges that when two or more activities interact with each other in a synchronized manner, it is very likely that the activities will wait for each other. Since waiting is expected to be a common occurrence, the mechanism has been optimized so that activities do not incur large overheads by being descheduled when they block as a result of a message operation. The STAROS operating system, which does not incorporate the optimizations made in MEDUSA, suffers from a descheduling overhead that is an order of magnitude larger than the cost of sending a small message whenever one of its processes blocks while performing a message operation.

A form of message communication that is very convenient is where an activity can wait for messages to arrive in any of a specified set of receiving ports. The MEDUSA microcode provides a **multi-event** mechanism that allows an activity to specify a number of receiving ports and then wait for a message to arrive in any one of them by executing a single operation. Another important feature of the communication mechanism in MEDUSA is that most of the message operations have been overloaded so that they can be applied not only to the objects that implement messages but to other objects types as well. The overloading provides a uniform way for activities to invoke ***Send*** and ***Receive*** operations without being aware of the details of how the operations are handled.

8.1.2.1. Messages

Message communication in MEDUSA is implemented using special objects called **pipes**. A pipe is a buffered, unidirectional communication medium that is typically shared by two or more activities that wish to communicate. Operations defined on pipes permit activities to send and receive data encapsulated in messages. MEDUSA pipes are similar to UNIX pipes in that they hold uninterpreted strings of bytes; they are different in several other respects. First, the integrity of messages in pipes is preserved by storing information about the size of each message in the pipe; when a message is received from a pipe the same number of bytes are removed as were entered when the message was put into the pipe. Second, the identity of the sender and the size of the message are made available to a receiver.

When activities communicate with each other using messages, both sending and receiving activities will normally encounter situations in which a message operation cannot be performed immediately. For instance, a ***Send*** cannot be done if the target pipe is full, and a ***Receive*** cannot be done if the target pipe is empty. The MEDUSA microcode provides both *conditional* and *unconditional*

forms of ***Send*** and ***Receive*** that correspond to the two ways in which such a situation can be handled. For the conditional versions the activity is notified that the operation failed. The activity is then free to perform the operation again or to do something else. For the unconditional versions, the activity is made to wait explicitly till the operation can be performed. The unconditional forms of ***Send*** and ***Receive*** are especially important because there is no way to emulate them using the conditional operations without incurring busy-waiting overhead. Furthermore, it is difficult to provide starvation-free service to each activity when a number of activities are performing conditional operations on the same pipe.

In a client-server relation between a group of activities, it is often desirable to organize the server activities so that each one of them can wait simultaneously on more than one request pipe. For instance, utility activities in MEDUSA are organized in this way for deadlock avoidance and efficiency reasons (this organization will be explained further in Section 8.3). One way to implement this form of waiting is to have each server activity poll its set of pipes continuously using a ***Conditional Receive*** operation. Since this busy-waiting wastes communication resources, a form of waiting that does not involve continuous overhead has been provided in MEDUSA. Each activity is allowed to specify a **multi-event set** of pipes from which it can receive messages from other activities. Pipes may be added to or deleted from this set dynamically using microcode operations. A microcode operation called ***Multi-Event Wait*** allows an activity to receive the first message that arrives in its set of pipes—while all of the pipes are empty the requesting activity is blocked. Thus ***Multi-Event Wait*** is logically equivalent to the ***Receive*** operation, except that ***Multi-Event Wait*** works with a number of pipes instead of just one pipe.

Since speed of the message mechanism is important to achieving good performance in MEDUSA, special care has been taken in the design and implementation of this mechanism to ensure efficiency. Two aspects of the design contribute especially to this efficiency: (a) the special treatment of the "pipe empty" and "pipe full" conditions and (b) the optimization of the ***Send*** operation when a receiver is waiting.

In a producer-consumer relation between two activities, it is unlikely that the producer and consumer will operate at exactly the same average speed. Thus even with the buffering provided by pipes either the sender or the receiver will normally have to wait. In most operating systems a process that is blocked waiting for some event to occur is automatically descheduled from its processor. Thus message transactions usually involve two context swaps, one when the process blocks and the second when the event that it was waiting for occurs. In MEDUSA we wanted activities to be able to interact on a substantially finer grain than that of the slow context swap on the LSI-11. To achieve this fine-grain interaction, an activity's state is *not* swapped off the processor as soon as the activity blocks. Instead, the activity is allowed to stay on its processor for a duration called the **paus-**

time that the activity may specify. If the activity unblocks before its pause time runs out, it is able to proceed immediately without incurring any context-swapping overhead; otherwise its state will be removed from the processor and another activity allowed to run. It is interesting to note that the role played by the pause time in making efficient communication possible is similar to that played by buffering. Buffering provides a way to smooth out *instantaneous* differences in the speed of the sender and receiver, but cannot account for *average* differences in speed. The pause time, however, takes care of average differences in speed by filling out the time at the faster activity, slowing it down till its average speed exactly matches the average speed of the slower activity.

Optimization of the ***Send*** operation on a "receiver waiting" condition directly affects the time it takes an activity to invoke operating system functions. A ***Send*** done by a requesting activity to the invocation pipe for the function will normally[17] find the pipe empty and the utility activity waiting for a message. In most implementations of message systems two transfers of data are performed regardless of the state of the pipe when the ***Send*** is done. In MEDUSA, data is transferred directly from the sender's buffer to the receiver's buffer if the receiver is waiting on the pipe. Besides speeding up the transfer, bypassing the pipe has the advantage that the pipe is locked for a much shorter duration than if data were transferred to and then from it. This allows sends done by more than one activity to be overlapped to a greater extent.

As a result of the two optimizations described above, the cost of invoking an arbitrary operating system function is the equivalent of executing 60 LSI-11 instructions (for actual measurements, see Section 8.1.6). This figure compares favorably with the 20 or so LSI-11 instructions that are required for a high-level language procedure call in which registers are saved and a display is updated. The equivalent cost in the STAROS operating system is over 300 LSI-11 instructions.

8.1.2.2. Events

Most of the pipe operations described in the previous section have been overloaded so that they can be applied to objects of other types as well. This overloading of pipe operations is similar in intent to the overloading of the *Read* and *Write* operations in the UNIX operating system. The motivation in both systems is to provide a mechanism for programs to communicate and to do I/O without being aware of what it is they are communicating with.

The way overloading is implemented in MEDUSA is to generalize the notion of an **event** to a number of object types in addition to pipes. Events are simply conditions that activities may wait for. For example, an event for a pipe is the presence of a message in the pipe; for a semaphore object an

[17] The number of activities in a utility is dynamically adjusted till each activity is slightly underloaded. This allows the utility to service requests more efficiently than if its activities were overloaded.

event is the condition that the semaphore integer is positive; for a file control block object an event is the presence of data in the buffer of the file control block. An activity may perform the overloaded pipe operations on any object type for which events are defined. If the event defined for the object has not occurred, the activity will be made to wait just as for a ***Receive*** on a pipe. If the event has occurred, any data corresponding to the event is transferred to a buffer specified by the activity, and the activity is allowed to continue. An activity may also put objects of several different types in its multi-event set and then wait for the first event to occur for an object in the set using the ***Multi-Event Wait*** operation. Object types on which events are defined are not restricted to be implemented in microcode; a number of the object types on which events are defined are, in fact, implemented by the utilities. When the microcode receives a request to perform one of the overloaded operations on a type that is implemented by one of the utilities, it converts the operation request to a call to the utility that manages the type and drops out of the operation.

8.1.3. Distributed System Issues

A number of the problems that had to be addressed in the design and implementation of the MEDUSA microcode arose directly as a result of the distribution and explicit parallelism present in the hardware of Cm*. Other problems, although present in more centralized systems, were harder to solve in the context of Cm*. The purpose of the next two sections is to outline these problems and to present the techniques used in MEDUSA to solve them.

8.1.3.1. Management of Descriptor Space

Descriptors for objects in MEDUSA are kept in descriptor lists that are distributed around the system. There are no constraints on the relative locations of an object and a descriptor list that contains a descriptor for that object.[18] Thus an object, a descriptor list that has a descriptor pointing to the object, and an activity that needs to reference the object may all reside in different clusters. This generality does have its price, however, since it makes the synchronization of multiple accesses to the descriptors for an object more difficult.

At any given time, there may be several microcode operations in progress that need to read a descriptor in order to access the object pointed to by the descriptor. Simultaneously, there may be other microcode operations that need to overwrite the descriptor with a new one. If only one copy of a descriptor were maintained (in its descriptor list), synchronizing these multiple accesses would be easy. The descriptor list could be used as the synchronization point and the problem could be solved

[18] This is in contrast to STAROS where a descriptor for an object must reside in the same cluster as the object itself. This limitation in STAROS also means that an object can never be moved outside the cluster in which it was created.

using the approach in [Courtois *et al.* 71]. However, for the sake of performance, descriptors need to be cached in Kmaps so there is usually more than one copy of a descriptor present in the system. Moreover, since the copies of the descriptor are distributed around in several Kmaps there is no one point at which synchronization can occur. As an added complication, any solution to the problem is constrained by the requirement that reads to a cached copy of a descriptor must proceed rapidly since they are in the critical path for object accesses.

The solution used in MEDUSA to synchronize concurrent reads and writes to multiple copies of a descriptor has two components. First, an up-to-date copy of the descriptor is maintained in at most two places: the descriptor list in main memory, which is always current and is assumed to contain the definitive value of the descriptor, and in the cache of the Kmap that contains the object. All other Kmaps contain indirect descriptors that provide the number of the Kmap containing the object but contain no other information.This Kmap number is treated by the system as a "hint" in the sense that it may be invalid at any time. When a request to access an object that is kept at another cluster uses a hint that is incorrect, it encounters a cache fault at the Kmap pointed to in the hint. This cache fault indicates that the hint is incorrect and causes the hint in the local Kmap to get restored from the copy of the descriptor kept in the descriptor list. The main effect of keeping only two copies of a descriptor is that the complexity of the descriptor read and write operations is considerably reduced.

The second component of the solution involves restricting all accesses to the two copies of the descriptor so that they occur in the same sequence: descriptor Kmap first, then the object Kmap. When a descriptor cache fault occurs at a Kmap, a request to read the descriptor is issued. This request travels first to the Kmap containing the descriptor to read out the descriptor. It then travels to the Kmap containing the object, overwrites any existing copy in this Kmap's cache and returns to the invoking Kmap with the indirect descriptor. When a request to write a descriptor is issued, the request follows the same route; it first updates the copy in the descriptor list and then updates the copy in the object Kmap's cache. Since inter-Kmap requests for service are guaranteed to arrive in the order in which they were sent, it is not possible for two or more requests to interact in such a way that an older copy of the descriptor remains cached permanently in the object Kmap.

8.1.3.2. Inconsistency, Starvation, and Deadlock

The three problems of inconsistency, starvation, and deadlock that arise in the context of allocating shared resources in a system are closely related to each other. Maintaining consistency of information that is shared in a read/write manner usually implies that the information must be accessed mutually exclusively by each requester. However, if a resource is accessed mutually exclusively, some arrangement needs to be made to handle requests that arrive when the resource is busy. Two approaches are usually taken to handle such requests. In the first approach, a report that

the resource is busy is sent to the requester who is then allowed to make the request again. In the second approach the requester is made to wait explicitly for the resource. The first approach needlessly consumes communication resources in busy-waiting for the resource to become available and leaves requests open to starvation.

The second approach avoids starvation but can lead to deadlock if two or more requesters make requests in such a way that each requester is waiting for another to release a resource before any of them can proceed. Note that although deadlock is also possible in the first approach, it *can* be avoided by simply releasing all resources held by a requester when it encounters a busy resource and then retrying. In the second approach, however, a requester cannot take any action when a busy resource is encountered because the requester is made to wait explicitly.

Although starvation is generally undesirable in a system since it leads to unpredictable behavior, it may be acceptable if the cost of avoiding deadlock is too high. Starvation cannot be tolerated in MEDUSA, however, because its presence would seriously compromise the robustness of the system.[19] Consequently, an operation is forced to wait explicitly if it needs to access a busy resource that can only be accessed mutually exclusively. FIFO queueing is used throughout to order multiple requests for the same resource since it guarantees fair service. This approach, however, leaves the system open to deadlock. Most classical solutions to the deadlock problem assume the existence of a single, central arbiter to resolve resource conflicts [Dijkstra 68]. A central arbiter cannot be assumed to exist in the context of Cm* unless one is willing to sacrifice both performance as well as reliability to guarantee freedom from deadlock.

The principal idea used to avoid deadlock in the microcode is to impose a systemwide partial ordering on all shared resources that microcode operations may have to wait on. The ordering of resources is static and was decided at system design time. All operations in the microcode follow this global ordering in acquiring resources during the course of their computation. Resources earlier in the ordering are acquired first and resources further down are acquired later. Since resource requests are ordered in this way, it is not possible for cycles to exist in requests for resources; consequently, deadlock over resource allocation is not possible. Note that the presence of a static, system-wide structuring of resources is sufficient to prevent deadlock even though both the resources to be allocated and the operations making the requests are distributed.

For this scheme to work in a straightforward way, an operation needs to know all of its resource requirements right from the start. Since this usually cannot be guaranteed, an operation must to be

[19] The reasons for this are too detailed to go into here, but have to do with the difficulty of discarding return messages from intercluster operation requests that have been timed out. This difficulty arises because of oversights in the implementation of the hardware.

able to abort processing when it discovers that it needs a resource that is earlier in the ordering than one it has already acquired. The operation must then be able to restart itself with the intention of acquiring the resources in the correct order.

As an example of how ordering is used to avoid deadlock, consider the allocation of Kmap contexts to a microcode operation. If contexts were allocated arbitrarily, it is easy to imagine situations in which circularities in requests for contexts could lead to deadlock (an example of such deadlock has been given in Section 4.4.3). As part of the deadlock avoidance strategy, the contexts in a Kmap are statically divided up into two classes called "master" and "slave" that are ordered for the purpose of allocation. In addition, all microcode operations are organized so that no more than two contexts (each perhaps in a different cluster) are needed simultaneously by an operation at any point in its lifetime.

When an operation starts out, it has no idea whether it will need two contexts or not, so it is allocated a context at random from one of the two classes. If the operation later discovers that it will need two contexts, but it has been allocated a "slave" context, it queues itself up to request a master context before trying to get a second context and releases the "slave" context that it holds currently. Once a "master" context has been allocated to the operation, it tries to get the second context from the "slave" class. Since all requests for contexts follow this discipline, and since "slave" contexts are guaranteed not to make requests for contexts to be allocated to them, deadlock over contexts is not possible.

8.1.4. Robustness and the Handling of Exceptional Conditions

Somewhat over one-fourth of the MEDUSA microcode is dedicated towards ensuring that the services provided by the microcode are robust. The term robustness is used in a fairly general sense to mean reasonable response by the system to changes that occur in its environment. The changes may or may not be errors, and could include hardware failures, abnormally heavy system loads, inconsistent data, or invalid requests for service. Robustness on the part of the microcode does *not* mean that all exceptional conditions are handled in such a way that their occurrence is masked from higher levels of MEDUSA. If an exceptional condition cannot be handled completely within microcode, it is considered acceptable to send a detailed report of the exception to the invoker of the failing operation.

Since the program-development tools available at the microcode level are poor and since microcode space is at a premium, only the exceptions that are simple to deal with or are otherwise crucial to reliable operation of the system are handled within the microcode. Other exceptions are simply reported to the invoker of the operation in sufficient detail to permit the condition to be handled

at that level. Most errors that are detected by hardware when a microcode operation is in progress are handled within the microcode by retrying the sub-operation that failed. If repeated errors occur, the condition is reported to the invoker after consistency is restored within the microcode.

A different aspect of robustness that involved much more complexity, but which was nonetheless considered important enough to implement in microcode is the response of the system when heavy load is placed on its components. The performance of a system under loaded conditions may be surprisingly different from that expected by its designers. Furthermore, subtle problems of data-structure consistency, and of starvation and deadlock over resources usually do not appear until contention for resources is present. Experience with earlier Cm* microcode systems had shown that problems of this kind are not infrequent, and are quite difficult to track down when they do occur. Considerable attention has been therefore been paid in the design and implementation of the MEDUSA microcode to ensure reasonable response under loaded conditions. Section 8.1.3.2 has described some of the techniques that were used to achieve this robustness, and Section 8.1.6 provides measurements which indicate that the system does respond reasonably to heavy load.

A complex microcode operation typically involves several Kmaps during its lifetime, and at any time there may be several contexts at different Kmaps computing on its behalf. There clearly needs to be some way to coordinate the actions of the various contexts involved in an operation when an error is detected by one of them. In the MEDUSA microcode this coordination is provided by structuring operations so that at any point during an operation there is a clearly defined context (the master context) that is responsible for reporting unrecoverable errors to the activity that initiated the operation. All other contexts are slaves. When an exceptional condition occurs that cannot be handled by a slave, the slave simply releases all of its resources and reports back to its master. The master takes care of aborting other slaves and reporting the error to the invoking activity. If an exceptional condition occurs that cannot be handled by a master, the master aborts all of the slaves and reports back to the invoker. Thus the master context for an operation provides a clearinghouse for reporting exceptions that occur while the operation is being performed.

A key idea in providing robust operation is *early detection*. If an error is detected as soon as possible after it occurs, the likelihood that this error will affect more than one locus of control and spread to other data structures is small. To ensure that errors are detected early and to avoid the use of inconsistent data, the microcode employs extensive consistency checking: approximately one out of every ten instructions in the microcode performs some sort of consistency check. The microcode is also assisted by the hardware in making checks on values placed on busses and read from memory locations.

Since exceptions occur rarely in practice, testing of code related to exceptions is more difficult

than testing most other portions of the system. The MEDUSA exception-handling code has been structured to alleviate this problem at least partially. Common portions of exception handling and exception reporting have been confined to a few well tested procedures that are used throughout the microcode. The code for detecting exceptions and invoking the appropriate procedures to handle the condition must unfortunately be distributed among all the procedures comprising the microcode. However, all of the complicated calling sequences and all of the detection checks that are used widely are encapsulated into a few in-line procedures to guarantee that each occurrence does not have to be tested individually.

8.1.5. Size Measurements—A Breakdown of the Complexity

The MEDUSA microcode is the largest and most complex microcode that has been written for Cm* to date. The total source code including comments amounts to over 20,000 lines, and the amount of object code to well over 4000 80-bit words. This section tries to account for some of this complexity and to identify portions of the microcode that were difficult to implement or turned out much larger than originally anticipated.

Table 8-1 shows the sizes of different portions of the microcode grouped by function and arranged according to decreasing size. The table lists the number of lines of source code, the number of lines of comments for that code, and the number of 80-bit microinstructions assembled from the source. Messages and events, even excluding the code that is responsible for the data transfer that occurs during message operations[20] constitute the largest part. Although the message mechanism is conceptually simple, it was difficult to implement in microcode. A significant portion of this complexity is a result of the optimizations that have been made to achieve efficiency; most of the remainder of the complexity results from the distribution of data and control present in the microcode.

The next largest part of the microcode consists of common routines that are used throughout the microcode. This includes routines to decode requests for service, routines to send intercluster messages, routines to manage the Kmap's data RAM, routines to perform commonly used type conversions, and routines to read and write words from main memory.

Another large fraction of the microcode consists of a collection of routines that, in one way or another, contribute to robustness of the microcode. In this collection are routines to queue and dequeue requests for critical resources, routines to handle hardware errors, and routines to report exceptional conditions to an activity. Exception detection and reporting code that is distributed throughout the microcode is not included under "robustness" in Table 8-1.

---

[20] *Data transfer is done using the block-transfer mechanism.*

Table 8-1: The Sizes of Various Portions of the Microcode

Function	Source Code (lines)	Comments (lines)	Object Code (80-bit words)
Messages and Events	1832	2538	901
Common Routines	1597	3469	750
Robustness	1312	2351	670
Address Mapping	1144	1950	610
Block Transfers	716	1343	367
Interrupt Handling	510	968	285
Activity Multiplexing	305	576	157
Indivisible Operations	217	344	105

The code that performs address mapping and implements shared memory is next in terms of size. One of the motivations behind the address structure of MEDUSA was to simplify this portion of the microcode. Experience with the first version of the STAROS microcode had shown that much of the complexity that was present in its implementation arose from two attributes: the generalized capability addressing structure of each process [Jones *et al.* 77], and the multiple levels present in the address-mapping process. Although the basic implementation of address-mapping in MEDUSA is quite simple, additional microcode did have to be written to handle the fact that address information is distributed around the system and no restrictions are placed on the relative locations of objects and descriptors for the objects.

8.1.6. Microcode Performance Measurements

This section provides performance figures for microcode operations that are used frequently by the operating system utilities and user programs. The figures are divided into two groups: the first group contains measurements taken on a lightly loaded system. Since there was little or no contention for resources under these conditions, the measurements provide an upper bound on the performance of the operations under more realistic conditions. The second group of measurements shows how the performance of three representative operations degrades when the number of activities performing the operations is increased until there is severe contention for resources. Taken together, these two

groups of measurements give a reasonable idea of the performance to be expected from the microcode under actual operating conditions.

Two things are worth emphasizing about the performance figures presented in this section. First, all data was obtained by making measurements on the running system, not by calculations based on the CYCLES program for tracing microcode. Actual measurements were used instead of traces because data from the CYCLES program does not account for queueing delays and contention in the hardware. Second, the performance figures correspond to a version of the microcode that has not been optimized after it was first implemented. Thus many of the key figures can be expected to improve considerably for an optimized version.

8.1.6.1. Measurements on the Underloaded System

This section contains times for the basic Kmap operations measured on a lightly loaded system. The time for each operation is specified in microseconds as well as in terms of the number of average LSI-11 instructions that can be executed in the same time. The second measure provides a machine-independent specification of the time taken to perform an operation, and is useful in comparing the times in MEDUSA to times for similar operations on other machines.

Table 8-2: *Microcode Operation Timings*—Simple Operations

Operation	Time (microseconds)	LSI-11 instructions
Load Window	69	9
Read Descriptor	63	8
Read Word		
(local cluster)	10	1
(non-local cluster)	30	4
Indivisible Increment		
(local cluster)	33	4
(non-local cluster)	47	6

Table 8-2 shows the timings of four of the simpler Kmap operations. The ***Load Window*** operation allows an activity to associate a descriptor in one of its descriptor lists with one of the sixteen windows in its processor's address space. Once a window is loaded, the activity may reference the object by going through the window. ***Read Descriptor*** enables an activity to read out information about the size, type, location and protection of the object from a descriptor for the object. The time in the table

assumes that the invoker and the descriptor list are in different clusters. The operation ***Read Word*** allows an activity to read a word from a page object, or from any other object if appropriate privileges are available. The times for both the intracluster and the intercluster case are given. ***Indivisible Increment*** is representative of a whole class of synchronization operations that allow an activity to manipulate read/write data indivisibly. The particular operation listed in Table 8-2 adds 1 to a given location.

Table 8-3: Microcode Operation Timings—Block Transfer

Number of Words	Time (microseconds)	LSI-11 instructions	Microseconds/Word
1	251	31	251.0
10	342	43	34.2
20	449	56	22.5
40	647	81	16.2
60	845	106	14.8
80	1075	134	13.5
100	1273	159	12.7
200	2295	287	11.5
2000	20728	2591	10.3

Table 8-3 shows the performance of MEDUSA's block-transfer mechanism. The time for a one-word transfer is 251 microseconds, or 31 LSI-11 instructions. Since the amount of data transferred is small, most of this time corresponds to the time required to set up the block transfer.[21] The table also shows how the time for a block transfer varies with the number of words to be transferred. It can be seen that that the mechanism becomes quite efficient for transfer sizes of 15 words or more. In particular, for transfer sizes larger than 15 words it is more efficient to block-transfer words from a remote cluster than to read them one by one using intercluster memory references. The asymptotic value of the transfer time for large blocks is 10.3 microseconds per word, which compares favorably with the figure of 5 microseconds per word that is the best that can be done because of hardware

[21] At the moment the block-transfer mechanism has not been optimized for small transfers—an optimized version could cut this time by about 60 microseconds for transfer sizes of less than 8 words.

limitations.

Table 8-4: Microcode Operation Timings—Message Operations

Operation	*Time (microseconds)*	*LSI-11 instructions*
Conditional Receive (empty pipe)	253	32
Send (one-word, non-full pipe)	341	43
Send With Bypass (one word)	480	60

The next set of measurements (Table 8-4) characterizes the performance of MEDUSA's message mechanism. Instead of listing the times for all of the message operations under a variety of possible conditions, a few key measurements were chosen. Those numbers that are not in Table 8-4 can be directly calculated from the ones given.

The first line of Table 8-4 gives the time taken to perform the ***Conditional Receive*** operation when the target pipe is empty (this measurement, as well as the others in Table 8-4 were made with the invoker and pipe in different clusters). Since no data is transferred under these conditions, the time is an estimate of the overhead involved in reading parameters for the operation, locating the pipe, checking its status, and notifying the activity. These actions are always performed for all pipe operations, thus the time involved is the minimum time any pipe operation can be expected to take.

The next measurement gives the time to send a one-word message to a non-full pipe using the ***Send*** operation. Since all data transfers in message operations are done using the block-transfer mechanism, the 341 microseconds includes the time to transfer a single word of data from the sender's buffer to the pipe. The time taken to perform a ***Receive*** of one word from a non-empty pipe is not shown in the table, but it is virtually identical to the time for a one-word ***Send***. The third line in Table 8-4 lists a key measurement for MEDUSA'S message system. It is the elapsed time from the moment a sending activity does a one-word ***Send*** to an empty pipe that has a waiting receiver to the moment the waiting receiver executes its first instruction following reactivation. *This is essentially the time it takes an activity to invoke an arbitrary operating system function in MEDUSA and is a fundamental time constant of the system.*

Table 8-5: Microcode Operation Timings—Privileged Operations

Operation	Time (microseconds)	LSI-11 instructions
Establish XDL	23	3
Reawaken Sleeping Activity	106	13
Write Descriptor	610	76

The last set of measurements gives the times for three privileged operations that are used frequently by the utilities. ***Establish XDL*** is used by a utility activity to map its auxiliary descriptor list[22] onto the PDL or SDL of another activity in the system. This ability to gain access to another descriptor list along with amplification privileges allows a utility activity to manipulate objects stored in another activity's address space. ***Reawaken Sleeping Activity*** is used to reactivate an activity when it is in the blocked state and has just become runnable. ***Write Descriptor*** is the operation used to change the contents of a descriptor indivisibly. This operation takes a fairly long time since it has to synchronize with other ***Write Descriptors***, ***Read Descriptors***, and Kmap contexts that are potentially using physical addresses derived from the descriptor to be overwritten.

8.1.6.2. Degradation of Performance under Load

The purpose of this section is to characterize the variation in the performance of the microcode as the amount of load on the system is increased. Toward this end, three operations ***Read Word***, ***Indivisible Increment***, and ***Conditional Receive*** were measured separately as the number of activities performing the operation was increased. The configuration of references was chosen to generate heavy contention in each case: for ***Read Word*** all references were made to the same memory location; for ***Indivisible Increment*** the same location was incremented by all the activities; for ***Conditional Receive*** the same pipe was used as target by all the activities. In addition, each activity was assigned to a different processor so that activities did not have to contend with multiplexing—each activity made operation requests at the maximum rate possible.

Figure 8-4 shows the measurements for each of the three operations. Although the operations covered a wide range of complexity, it is interesting that the shapes of the three curves are essentially

[22] The auxiliary descriptor list is a third descriptor list in the virtual address space of an activity. It is used only by utility activities.

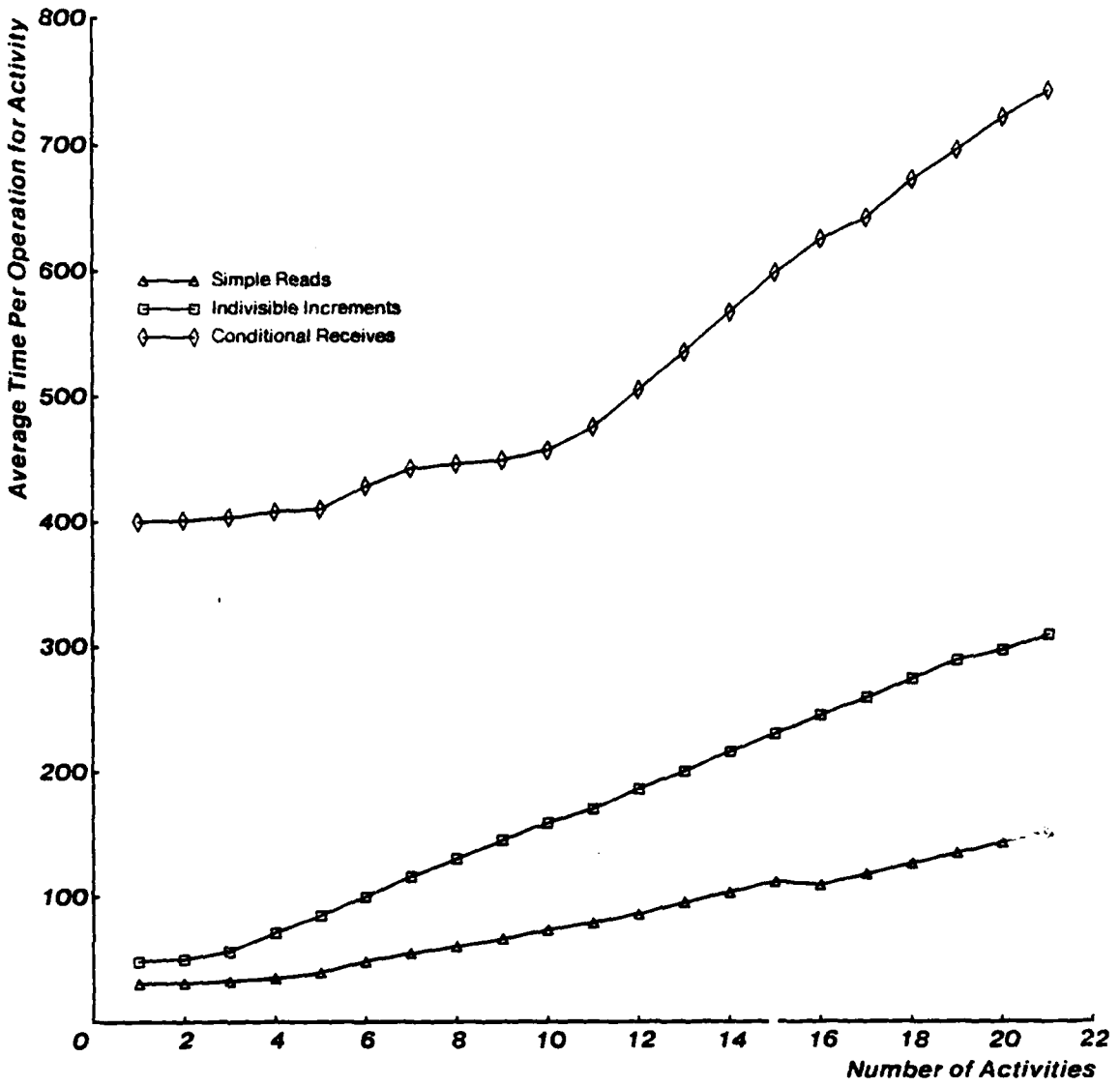


Figure 8-4: Degradation of Three Operations Under Load

the same. There is an initial portion in which the time to perform the operation degrades slowly, and a final portion in which it degrades more rapidly but the degradation is still linear in the number of activities. Since the operations ***Read Word***, ***Indivisible Increment*** and ***Conditional Receive*** are fairly representative of the set of microcode operations as a whole, other microcode operations can be expected to perform similarly.

8.2. The MEDUSA Kernel

The kernel is the portion of MEDUSA that is responsible for providing low-level device-handling code and for multiplexing activities that reside on a given processor. Each processor in the system contains a copy of the kernel, which consists of about 500 words of basic code and 250 additional words for each device type on the processor. Since the kernel is implemented below the level of utilities, and since its code is duplicated in every processor, there are good reasons for keeping down the size of the kernel. Thus the kernel provides only the bare minimum of facilities that need to be present at each processor.

The kernel issues commands to I/O devices and responds to interrupts generated by them. It chains together commands for devices so that commands can be buffered ahead to improve the throughput of high-speed devices. Interrupts generated by I/O devices are converted by the kernel into messages that are sent to the appropriate utility.

Each processor can contain up to 16 activities that are multiplexed by the kernel. In performing the multiplexing, the kernel makes no decisions about which activity on the processor is to run next; it simply provides the mechanism to make the activity switch.[23] Policy decisions are made at a higher level by the task force manager utility.

8.3. The MEDUSA Utilities

Operating system functions such as memory management, I/O, and file management are handled by five utilities. Each utility is a task force that contains several activities and implements a particular abstraction for the rest of the system. Parallelism is therefore built into the lowest levels of the operating system. The design of the utilities is such that the number of activities in a utility may change dynamically in response to changes in system load. However, for reasons of robustness each utility always has at least two activities.

The boundaries between utilities are rigidly enforced and are crossed only by messages. Communication among utilities as well as between users and utilities occurs via a set of pipes that is reserved for this purpose. Every processor in the system contains a special descriptor list called the **utility descriptor list** (UDL) that contains descriptors for the set of communication pipes. When an activity wishes to invoke a function implemented by one of the utilities, it sends a message to the

[23] This is an instance of the principle of separating policy from implementation that was first formulated in HYDRA [Levin *et al.* 75].

invocation pipe corresponding to the function. The ***Send*** operation used is exactly like a regular pipe send except that it takes a UDL slot as argument instead of an arbitrary pipe descriptor. The correspondence between slot numbers in the UDL and functions in the utilities is uniform across all UDL's in the system and was assigned statically at system design time.

Distributing the descriptors for communication pipes among UDL's that are kept one per processor has several advantages. Loss of a particular UDL has very local repercussions since only the processor containing the UDL will be unable to communicate. Other portions of the system will be not be affected. The ability of utility activities to share the workload for a particular function is also enhanced. Requests for a function may be redirected to a different activity on a processor-by-processor basis simply by overwriting the descriptors corresponding to the function in the appropriate UDL's.

A result of the distribution of functions between utilities is that circularities in function invocations between utilities could lead to deadlock over resources in a particular utility. Since each utility depend on other utilities to perform its own functions, circularities between utilities as a whole cannot be avoided. The solution used in MEDUSA is to divide the functions provided by each utility into a number of service classes so that (a) a given service class is completely implemented within one utility and (b) there are no circular dependencies between the service classes. This is a straightforward generalization of the solution described earlier to avoid deadlock over contexts in the microcode. In the microcode solution, contexts are analogous to functions in the utilities.

Given that the functions provided by each utility are divided into a number of service classes, these service classes must be mapped onto the activities of a utility in some way. A simple approach is to let each activity provide the services for exactly one class. Unfortunately, this approach has a number of disadvantages that make it unacceptable. Since some service classes will rarely be invoked, the activities assigned to service these classes will be poorly utilized. In addition, load-sharing between activities responsible for different service classes is not possible. Finally, since all activities of the utility are not identical, they will not be able to help each other in the event of failure. The approach used in MEDUSA is to let each utility activity provide all of the service classes implemented by the utility. Each activity is provided with its own collection of invocation pipes, one pipe per service class. The activity multiplexes itself between the various service classes by performing the ***Multi-Event Wait*** Kmap operation.

8.3.1. The Memory Manager Utility

The memory manager and Kmaps together provide for the management of the primary memory of Cm*, as well as for the implementation of page, pipe, and semaphore objects. The functions of the

memory manager fall into three classes: (a) allocation and deallocation of primary memory; (b) management of descriptor lists; and (c) initialization of page, pipe, and semaphore objects.

The memory manager maintains the allocation state of the main store. Other utilities must make allocation requests to the memory manager when they wish to create new objects. The memory manager also has the responsibility of returning storage to the free pool when it is no longer in use. MEDUSA uses a reference-count scheme to determine when the last descriptor for an object has been deleted, and the memory manager automatically frees up the storage allocated to the object when this happens. STAROS, on the other hand, uses garbage collection to retrieve storage that is no longer in use. The principal advantage of MEDUSA's scheme is that it is much more dynamic compared to garbage collection. Storage is reclaimed immediately after it becomes available instead of at the next time the garbage collector runs.

The second group of functions provided by the memory manager is used to manipulate descriptor lists. The memory manager is responsible for both the creation and deletion of descriptor lists. Operations are provided to move and copy descriptors between slots in the same or different descriptor the lists, and to reduce the privileges in a descriptor.

The third group of functions provides for the initialization of page, pipe, and semaphore objects. For the sake of efficiency, all operations on these objects except creation and deletion are implemented by the Kmaps. Since speed of creation and deletion are not as important, these operations are implemented in the more comfortable environment of the memory manager.

8.3.2. The File System Utility

The file system utility acts as a controller for all the input and output devices of the system. It implements a hierarchical file system that is based on that of UNIX but extended in several ways. The restriction on the small number of open files per process in UNIX is removed. UNIX allows a process to specify at most 16 file descriptors at any time. In MEDUSA, a descriptor for an open file is kept in one of the activity's descriptor lists; potentially there may be as many open files as there are slots in the descriptor lists.

A search-list mechanism is implemented in MEDUSA that allows several "current directories" for each activity instead of the single current directory for each process in UNIX. Finally, the organization of data on disks has been modified to make it more robust in the face of system crashes. The techniques used to achieve this robustness are quite similar to the ones used in the ALTO file system [Lampson and Sproull 79]. Each file is organized into blocks of fixed length. A label that contains the name of the file to which the block belongs, the number of the block within the file, and pointers to adjacent blocks is written into each block. When a file system crash occurs, labels provide a means

of reconstructing all of the information that is needed by the file system during normal operation.

Several interesting issues arise because of the distribution of the file system utility. When a file system activity wishes to perform an I/O operation, it is unlikely that the processor containing the I/O device will be the same as the one containing the activity. Communication of requests for I/O operations is carried out by providing a device-interface object that is shared by the kernel of the processor containing the device and the file system activity. This object contains a queue of pending requests for the device and other state information about the device. Communication in the other direction occurs via messages. The device routine mechanism enforces a clean separation between functions provided at the interrupt level and those done at background level.

8.3.3. The Task Force Manager Utility

The task force manager utility creates, schedules and deletes task forces. An existing task force may create a new task force by specifying the name of a task force description file to the task force manager. Parameters may be passed to the new task force as descriptors or as data that is pushed onto the stack of the first activity that gets created.

An existing task force may increase its size by adding new activities. A new activity is created by specifying the location and size of the private descriptor list for that activity, and by indicating descriptors to be placed in the new descriptor list.

Information about the state of a task force is kept in its task force control block. The task force control block contains information such as the name of the task force, the names of all the activities in the task force, and accounting information relating to CPU and file system usage. Ownership of a descriptor for a task force control block enables an activity to perform debugging operations on the task force, subject to the privileges contained in the descriptor. Operations include halting, restarting, and deleting activities of the task force.

One of the most important functions performed by the task force manager is to provide a scheduling policy that guarantees that all the activities of a task force will execute concurrently. This notion, termed **coscheduling** in MEDUSA, is crucial for good performance if the activities of a task force need to interact frequently. If the activities in such a task force are not coscheduled, some of the activities in the task force will normally not be scheduled on their processors because they are blocked waiting for results from other activities. When one or more of the running activities needs to interact with one of the descheduled activities, it too will block and will lose its processor. When a descheduled activity gets back on its processor there is no guarantee that the next activity it will interact with is not descheduled. As a result the activity may get descheduled soon after it has been scheduled. This is a form of thrashing that is similar to the thrashing that occurs in demand-paged

systems when the set of pages available to a process is much smaller than the set of pages the process needs.

8.3.4. The Exception Reporter Utility

This utility, in conjunction with the Kmaps, is responsible for the reporting of exceptions. In MEDUSA, the reporting of exceptions and their detection are separated. Whenever an exception is detected, either by the hardware, or by microcode in the Kmaps, or by one of the utilities, the exception reporter is notified. The exception reporter therefore provides a clearinghouse for all exceptions generated in the system, and is responsible for directing each exception to the handler that has been defined for exceptions of this type.

Since the reporting of exceptions is encapsulated in a single utility, it is easy to provide a uniform reporting scheme that applies to exceptions regardless of their origin. The implementation of the mechanism is completely hidden from other utilities: they use the reporting mechanism just like user activities. The conceptual centralization of the implementation makes it less likely that there will be subtle problems of interaction with details of the utilities.

8.3.5. The Debugger and Tracer Utility

The notion of debugging and measuring task forces has been formalized in MEDUSA in the form of a separate utility called MACE. This utility consists of a single activity, all of whose resources are allocated to a single processor. MACE executes with utility privileges and uses the standard MEDUSA message system for the purpose of communication. However, MACE does not depend on the facilities provided by the other utilities in any way, nor are the local resources used by MACE shared with other utilities.

Dedicating a processor to MACE and making it independent of the other utilities has several advantages. First, crashes in the rest of the system are not likely to bring MACE down. Second, the full processing power of an LSI-11 is available for use in measurement, without interference that might perturb the measurements. Third, MACE can be used to debug the utilities since MACE itself does not rely on utilities.

The facilities provided by MACE include setting of breakpoints and tracepoints in activities, symbolic examination and modification of the code of activities, and calling of subroutines.

8.3.6. Utility Performance Measurements

Table 8-6 shows the times for a number of the utility calls provided by the memory manager, the exception reporter and the file system.[24] Each figure gives the total time from the initiation of the operation to its completion when the user program is reactivated. In each case the pause time for the invoking activity is large enough that the activity will not lose its processor while it is blocked waiting for the call to complete.

Table 8-6: Utility Operation Timings

Operation	Time (miliseconds)	LSI-11 instructions
Utility Entry + Exit	2	250
Create Page + Delete Page	18.5	2300
Create Pipe + Delete Pipe	23	2900
Copy Descriptor	29	3600
Read Memory Statistics	14.5	1800
Read Disk Block	58	7300
Write Disk Block	37	4600

The first entry in Table 8-6 indicates the time for a null utility operation. The null operation simply parses the invocation message and returns immediately. Utility entry/exit calls are analogous to kernel or monitor entry/exit calls in more centralized systems. For example, kernel entry/exit in HYDRA is about 150 instructions [Wulf *et al.* 74]. Estimates for both UNIX and TOPS-10 are between 100 and 200 instructions.

The next two measurements of the table show the times required for utility calls that overwrite (indivisibly) a descriptor for a page or pipe object with a descriptor for a new object of the same type. The times for the operations include allocating memory for the new object and releasing the memory used by the old object. The ***Copy Descriptor*** time is for a utility call that overwrites one descriptor with another descriptor for the same pipe (thus no memory reallocation occurs). ***Copy Descriptor*** is somewhat more expensive than ***Create Pipe + Delete Pipe***: the greater cost is due to additional

[24] The material in this section has been taken directly from [Ousterhout 80].

synchronization overheads (two descriptors must be locked instead of one) and manipulation of backpointers. The ***Delete Null Descriptor*** operation returns immediately once it is known that the descriptor being deleted is null; this time measures the overhead in a utility to parse the invocation message (which contains the command and one additional parameter) and return a result. Each of these calls involves only a single utility, the memory manager.

The last two lines of Table 8-6 are for file system operations that require information to be read from or written to disk. Each figure is the average time for 1000 consecutive disk operations. In the case of ***Read Disk Block***, no buffering ahead is done by the file system (at the time the user activity invokes the file system the desired block is not in memory). Most of the time is spent waiting for the disk to transfer the block to memory. ***Write Disk Block*** is organized to buffer requests ahead: the operation completes as soon as the block has been queued for transfer to the disk. Since the disk is the bottleneck in ***Write Disk Block*** and the time in Table 8-6 is an average over 1000 consecutive writes, the 37-millisecond time per operation represents the disk transfer time, including seek time, rotational latency, and transfer time. Since the disk transfer times are identical for read and write, 21 milliseconds of the 58 total required for ***Read Disk Block*** is due to execution time in the file system utility; the rest is disk time.

Table 8-6 suggests that the performance overheads incurred because MEDUSA is distributed are negligible for utility calls. The cost of even the simplest utility call is around 1000 average instructions, whereas the cost of a message transfer is only about 60 instructions. Most of the time for a utility call is spent in parsing the call parameters and performing the operation; this overhead must be incurred by any operating system regardless of whether it is distributed or centralized. Tables 8-4 and 8-6 indicate that the message mechanism is efficient enough to support much finer grain utility calls without becoming a performance bottleneck.

8.4. Conclusions and Status

This chapter has provided an overview of the design of the MEDUSA operating system and a discussion of some of the motivations that led to this design. It has also provided preliminary measurements of key portions of the operating system to help characterize the performance that is attainable by programs running on MEDUSA.

One of the original goals of the MEDUSA project was to determine if a highly distributed structure like that of MEDUSA had a performance that was close to the performance that could be attained by programs running on the bare machine. The performance measurements indicate that the price that has to be paid for the distribution present in MEDUSA is negligible: most of the cost of operating

system functions is in performing actions that would be performed regardless of whether the system is distributed or not.

At the time of writing of this report, MEDUSA was almost fully operational. All of the microcode, kernel code and the utilities had been debugged to the point where several programs had been able to use the system successfully. Portions of the system, however, still need work before MEDUSA can be used routinely for the purpose of experimentation. In particular, a user version of the debugger and tracer needs to be available. The performance measurements made on the system provide only a rough idea of how well MEDUSA will perform in practice. Much more experimentation needs to be done to evaluate the dynamic behavior of the system and to determine whether real application programs can effectively utilize the hardware of Cm* using MEDUSA.

Future plans for the MEDUSA system include bringing up a UNIX shell and several of the UNIX programs in order to provide a reasonably hospitable environment on Cm* itself. There is considerable hope that this can be achieved easily, especially since the file systems of the two systems are almost identical. Most of the work involves writing runtime routines in the C programming language that would translate UNIX operating system calls into MEDUSA calls. The UNIX effort on MEDUSA has already shown some results. We have been able to execute simple programs written in C on MEDUSA; the UNIX shell and most of the UNIX utility programs are expected to be brought over in the near future.

8.5. References

[Courtois *et al.* 71]
P. J. Courtois, F. Heymans, and D. L. Parnas.
Concurrent control with readers and writers.
Communications of the ACM 14(10):667 – 68, October, 1971.

[Dijkstra 68] E. W. Dijkstra.
Co-operating sequential processes.
In F. Genuys (editor), *Programming Languages*, pages 43 – 112. Academic Press, New York, 1968.

[Jones *et al.* 77] A. K. Jones, R. J. Chansler, Jr., I. Durham, P. Feiler, and K. Schwans.
Software management of Cm*—a distributed multiprocessor.
In *National Computer Conference, Proceedings, Vol. 46*, pages 657 – 63. AFIPS, 1977.

[Lampson and Sproull 79]
B. W. Lampson and R. F. Sproull.
An open operating system for a single-user machine.
In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 98 – 105. ACM/SIGOPS, Pacific Grove, California, December 10 – 12, 1979.

[Levin *et al.* 75] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf.
Policy/mechanism separation in Hydra.
In *Proceedings of the Fifth Symposium on Operating Systems Principles*, pages 132 – 40. ACM/SIGOPS, University of Texas at Austin, November 19 – 21, 1975.

[Ousterhout *et al.* 80] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu.
Medusa: an experiment in distributed operating system structure.
Communications of the ACM 23(2), February, 1980.

[Ousterhout 80] John K. Ousterhout.
Partitioning and cooperation in a distributed multiprocessor operating system: Medusa.
PhD thesis, Carnegie-Mellon University, April, 1980.

[Wulf *et al.* 74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, F. Pollack.
Hydra: the kernel of a multiprocessor operating system.
Communications of the ACM 17(6):337 – 45, June, 1974.

9. ECHOES

Mike Kazar

And so I throw the windows wide
And call to you across the sky
– Pink Floyd from "Echoes"

ECHOES is a small experiment in inter-module communication, coded as a kernel of an operating system on Cm*. ECHOES has two goals. The first is to implement in microcode a fast call and return mechanism between protected subsystems. The second is to extend this procedure-call mechanism naturally so that instead of doing the procedure call within the same process, the procedure call may be done in a *new* process *in parallel* with the continuation of the calling procedure.

This is implemented by providing a collection of 28-bit address spaces in which execute an arbitrary collection of processes. Transfers from one address space to another are accomplished by means of ***Call*** and ***Return*** operations. Processes are created by means of a special version of ***Call***, known as ***Call/Fork***, which functions similarly to ***Call***, except that it has the ability to create a new process to be created to execute the called procedure.

An important property of ECHOES that should be noted is that address spaces are supposed to be independent of the processes that execute in them. Thus an address space can have zero, one or more than one process currently executing within it. The address spaces that these processes see are identical with one another with the sole exception that each process has its own stack to prevent interference with other processes in the same address space.

Cm* is an ideal test vehicle for an implementation of this system. There are two main reasons for this.

- We can actually run the called procedure in parallel with the continuing execution of the caller, rather than just simulating this parallelism, since we have more than one processor to work with. This is useful since it allows actual measurements to be performed on an actual running system. In addition, building a running system provides more assurance that no major difficulty has been inadvertantly overlooked.

- We can modify the addressing architecture of the machine arbitrarily by writing Kmap microcode to interpret any or all of the memory references made by the LSI-11's.

9.1. Some Properties of ECHOES

ECHOES has taken the approach of separating out the large, expensive addressing environment usually associated with a process from the rest of the process state. The creation of a new address space is considered to be a relatively expensive operation, while on the other hand, creation of a new process is a relatively inexpensive operation (that is performed in microcode). This stands in marked contrast to most operating systems, where creation of a process, especially on a machine providing a large virtual address space, is an extremely expensive operation. It is achieved by allowing an address space to persist even when a process is no longer executing in it, and perhaps later to be re-used by that process or another process.

In ECHOES there is a new address space for each user's instantiation of a subsystem. Processes are created dynamically to run in this address space when the particular subsystem is invoked (by ***Call*** or ***Call/Fork***).

Where multiple invocations of a subsystem are made the subsystem may construct an arbitrarily rich execution environment in own storage which persists across invocations because it is part of the address space. Consider the implementation of an editor which is to be implemented as a protected subsystem, *i.e.*, it can not do any damage to those who call it, nor damage those subsystems that it calls. Assume that when we call the editor utility to perform some work for us, we pass it two arguments, a string to edit and a place to put the resulting edited version of the string. Further assume that we want the editor to maintain state between executions. For instance, we might want it to remember the set of macros that we are using, a personal profile describing how the editor is to act in those areas where it provides a choice (*e.g.*, should we ask for confirmation before writing out a file), and other pieces of environmental information such as the list of text buffers begin edited.

To attain an efficient implementation of the editor, two distinct goals must be met. The first is to be able to pass *large* arguments between subsystems efficiently. The second is for a large environment to be maintained by a subsystem *between* invocations of it.

There are two types of systems that try to solve these problems. They are categorized here by their ability to pass a name for an object from one environment to another environment (such as from one process to another) efficiently.

1. *Systems without the ability to name objects across contexts.* These systems have a great deal of difficulty passing large arguments around efficiently. This is because there is no name for the argument in the called procedure's address space; thus the argument must be actually copied into the called procedure's address space.

2. *Systems with the ability to name objects across contexts* (*e.g.*, capability-based). With

these systems, one can build an environment object which one then passes as a parameter with each call to the subsystem's server process. Since an arbitrarily complex object (or set of objects) can be passed with each call at little cost, these systems can achieve the same ability as ECHOES to have a protected subsystem provide a function to an individual user in the context the user desires.

Next we discuss the question about how the subsystem is to maintain state between invocations. One method used in some capability-based systems is to simply send a complex object describing the state to a process which provides the protected subsystem's service. This, however requires a bit of work on the part of the caller, who must get this state object created by the called subsystem, and pass it off to the proper procedures of that subsystem when making the calls, which implies that the user must know what subsystem each procedure being called is a part of.

The alternative that ECHOES implements is fairly complex. Associated with each user and each protected subsystem is an address space. When a process running under ECHOES wishes to invoke a particular subsystem, rather than sending a message to an already existing process, including as part of that message a state-describing object, ECHOES creates a new process which (with the exception of having its own stack segment) executes in the address space associated with the protected subsystem and the particular user. This process creation is relatively inexpensive because the most expensive part, its address space, has *already* been created. One advantage of this method is that the caller of a subsystem need not know which state-describing object(s) to pass to a particular routine.

Note that ECHOES is not really a capability-based system. More specifically, it passes segments and sections of segments around from process to process, but most certainly does not allow users to copy these segment descriptors out of the database describing an address space. It maintains a single object type, called the **segment**. Sections of a segment are also sometimes utilized. For example, the ***Call*** operation can actually pass parts of segments around as arguments, so that for example a single word in the local stack could be passed as a parameter to a protected subsystem without fear that the rest of the stack could be damaged if that subsystem were to run wild or be maliciously trying to confuse its caller.

9.2. Definitions and Basic Operations

In this section I will present the essential definitions required to understand this paper and brief definitions of the most important operations provided by the ECHOES kernel.

- **Address Space.** A collection of 256 segments that are addressable at one time from the ECHOES virtual machine. For each segment in the address space, there is a vector of protection bits specifying the access a process using this address space has to each individual segment. One slot in the address space is not really a part of the entity called

an address space. Instead of containing a specific segment all the time, it contains the stack segment of the process that is executing in that address space. Thus if there are two processes executing in one address space, then the addresses that they generate refer to exactly the same objects, with the exception of the stack segment. References to the stack segment's addresses cause references to the process' own private stack object.

- **Protection Domain**. A set of ordered pairs of the form *(segment, access privileges)*. Among the possible access privileges are read, execute and write access. For each user, there is a separate address space created for each protection domain being utilized. Every address space in ECHOES has associated with it a protection domain. The access that a process has to the segments in a particular address space is that access which is specified by the protection domain associated with that address space.

- ***Call***: This operation is used by a process to transfer execution to a procedure it wishes to invoke. If the procedure is in another address space (and thus part of another protection domain) the ***Call*** operation will manage the transfer of control between address spaces too.

- ***Call/Fork***: This operation is used in the case where a ***Call*** operation is desired, but the caller wants the called procedure to execute in a separate process—*i.e.*, in parallel with the caller, if possible. If the called procedure is a part of the same protection domain as the caller, then the called procedure will run in the same address space as the caller, except for having its own stack segments.

- **Global Name**. A global name is a name that is unique within the entire system. It thus refers to the same object no matter which address space or process the name.

- **Job**. A job is a collection of processes and address spaces, which can trace their existence back to a common *initial* process. At a given moment, each process is executing in exactly one address space. Typically, the processes are working together to accomplish some common goal for some user. A job is quite similar in this respect to a Task Force in STAROS.

- **Process**. Basically, a process is a repository for the state of a virtual processor. Such things as the machine registers, the machine's pointer registers (special 32-bit registers maintained in the Kmap) and the program counter are part of the process state. The process has an associated address space in which the process is executing. However, the address space is not really part of the process; it is instead a separate entity that can exist independently without regard to the existence of any processes that are using that address space.

- ***Return***: The ***Return*** operation is used to return from a ***Call*** or a ***Call/Fork*** operation. If a return is made from a ***Call/Fork***, then the process executing the return is terminated, and a semaphore in the caller's address space is ***V***'d so that the caller can tell that the called process has completed. This semaphore is ***P***'d when the caller does a ***Synchronize*** operation in order to wait for the subprocess to complete.

- **Segment**. A collection of data, of size up to 1 megabyte, consisting of up to 1024 pages, each of size 1024 bytes. Segments are the basic data-containing objects in the ECHOES system.

9.3. The Architecture

The ECHOES architecture basically consists of many large (28-bit) address spaces, with a collection of processes executing in some subset of these address spaces and transferring among these address spaces during their execution histories. Two or more processes can execute simultaneously in the same address space, except that they will have their own private stack segments.

More precisely, each process executing in ECHOES sees a collection of 28-bit address spaces, between which it can transfer via the inter-procedure call mechanism. At any one instant a process can only refer to segments in its "current" address space.

These address spaces form the basis for the protection mechanism used by ECHOES. The access that a process has to a particular segment is a function of the *address space* in which the reference to the segment is made. Thus these address spaces are the primary mechanism used in the implementation of the protection domains.

The Kmap also provides support for sixteen 32-bit index registers (hereafter referred to as **pointer registers**). These registers can be used for indexing or data storage, just like normal LSI-11 registers. Their actual implementation is beyond the scope of this paper. They are used when addresses longer than 16 bits are needed. The LSI-11 architecture has been extended with assistance from the Kmap so that these index registers can be used in place of or along with regular LSI-11 registers. For example, in ECHOES the instruction

```
ADD 0(PR1),0(R2)
```

is valid and means that the processor should add the contents of the word at contents(PR1) to the contents of the word at contents(R2) and put the result in the word at contents(R2).

The Big Picture of an Address Space. This section describes segments in an ECHOES address space. Segments come in three flavors.

1. *Per-Address Space.* The most common type of segments are those that occur on a per-address space basis. This means that no matter who is executing in this address space, these segments are always in that address space in their appointed places.

2. *Per-Process.* There are some segments that really are per-process databases. One example of such a segment is the user's stack segment. Since the basic idea is to allow multiple processes to come into existence in the same address space when the need for the processing power presents itself, it is necessary to ensure each of these processes has its own private stack segment so that two such processes do not interfere with each other.

3. *Per-Call.* The last flavor of segments is per-call segments. These are segments that contain the arguments when a procedure is executing. Whenever a procedure is called it

executes in an environment where its arguments are mapped into eight reserved segments. These segments are clearly not per-address space or per-process since one can perform a ***Call*** and stay within the same address space or within the same process.

Now we are ready to explain how the 256 segments of an address space are used. It is important to note that here we are talking about the uses of addresses, not particular virtual memory objects in the system.

- *Segments 0 – 3.* These segments never contain any data. Referencing them causes a fault. They are reserved for a large address space of addresses that generate faults when referenced, and so that small integers (those in segment 0) generate faults that the user will see as errors if these addresses are indirected through. They are empty slots in the address space.

- *Segment 4.* This segment is the gate segment. It is used by the call mechanism to provide a way of specifying how to call a procedure in a different address space. The details of this mechanism are beyond the scope of this report, but the *flavor* of this segment is per-address space.

- *Segment 5.* This segment is the protected call stack used by the microcode for the implementation of the ECHOES virtual machine. Each ***Call*** pushes a small stack frame on this stack, and ***Return***s pop the same frame. Only the microcode can successfully reference this segment. It is a per-process segment. Note that if two processes are executing in the same address space, segment 5 is different for them.

- *Segment 6.* This segment is the process' stack. Frames on it are pushed by the ***Call*** operation too, but of course the user's stack is accessible by the user's process. However, information on the stack pushed before a procedure call is *inaccessible* to the called procedure. Note that if two processes are executing in the same address space, segment 6 is different for them.

- *Segment 7.* This segment is the descriptor segment. It contains the definition of the address space. It is, of course, of the per-address space flavor. Basically, a descriptor segment specifies for each segment's slot in the address space the unique ID of the segment that goes there, and the access that is had to the segment when running in that address space.

- *Segments 8 – 15.* These segments are the argument segments. After a procedure call, segment $8+i$ contains the ith argument. Furthermore, the access that the called procedure has to this segment is the intersection of the rights the caller had and what the caller desired to pass on to the called procedure. These are per-call flavor segments.

- *Segments 16 – 255.* These segments are the segments in the address space that are available for people to use for data, programs, etc. These are per-address space segments.

When a user first calls a procedure that implements part of a protected subsystem, if an address space corresponding to the protected subsystem's protection domain doesn't already exist in this job, then it is created. The segment containing the procedure is placed in the newly created address

space. Thus when this procedure is executed, it will have the proper access rights to segments in its address space, *i.e.*, the same rights as other procedures implementing this protected subsystem for the user.

9.4. Operations

There are two important types of operations that can be performed by the system. The first is a *simple procedure **Call***. The simplest scenario here is the case where no arguments are passed. What happens is that the microcode remembers, in a stack frame pushed on a stack used only by the microcode, the program counter and address space from whence the ***Call*** was made. This stack is contained in segment 5 in the address space. The process then switches to the new address space and changes the program counter to its new value. When a ***Return*** operation is performed, the program-counter/address-space pair on the top of this stack is popped off and restored into the current process state.

In the case where arguments are passed to the procedure, before the ***Call*** we specify via microcoded special instructions the location, size and access privileges (subset of the process's access rights to the argument in its current address space) for each argument. When the ***Call*** is actually performed, it sets up the new address space so that the first eight arguments appear in eight reserved segments in the new address space (segments 8 through 15 respectively). These eight segments are refered to as argument segments. The segments containing the called procedure's arguments appear in the new address space with the properly restricted access rights, and also with restricted size (*"windowed"*) so that the *only* part of the argument segment that is accessible is the region starting at the first word of the argument and continuing for the exact length of the argument, as specified by the caller. The resolution of this protection is one PDP-11 word. The ***Call*** operation additionally sets (Kmap-maintained) index registers 8 through 15 to point to the base of the arguments, so that it is easy for the called procedure to reference its arguments. This however puts an effective limit on the number of arguments that can be passed to a procedure, but it has not been a problem so far. Note that this mapping of the segments containing the arguments occurs even when the ***Call*** is into the same address space as the caller's. It is important to note that within a particular address space the argument segments seen by two different processes are different, just as the two processes see different stack segments even when executing in the same address space.

The ***Call/Fork*** extension to this operation is fairly simple. When a ***Call/Fork*** operation is done instead of a ***Call***, the system has the option of allocating a new process to handle the called procedure. In addition, the system provides a means of testing for completion of the called procedure's process. Currently this wait operation uses busy-waiting to wait for the other process,

but there is no reason why it could not simply block the waiting process and cause the scheduler to be invoked.

The ***Return*** operation's definition has also been augmented, so that if it returns from a ***Call/Fork***, then a binary semaphore in the caller's address space is *V*'d and the ***Call/Fork*** process is terminated.

9.5. An Example of Using ECHOES

In this section we will step through a typical ECHOES call and return sequence, for a procedure running in a separate protection domain from its caller that takes two arguments (*a*,*b*) and adds *b* to *a*. We will assume that both arguments are one LSI-11 word long.

The caller first performs two ***Load Arg*** microcode operations, which specify the address, size and access-rights restrictions of the arguments. The first one specifies all the rights that the caller had to its argument, the address of *a* and a size of 2 (bytes). The second ***load arg*** will specify the address of *b*, a bit map restricting write access (since the procedure does not need to be able to write in *b*), and also a size of 2. Then the ***Call*** is performed. Segment 8 in the new address space is mapped over the segment containing *a*, and similarly segment 9 is mapped over the segment containing *b*. However, references to any location in these segments except for the one word containing the argument will result in an out-of-bounds fault. Furthermore, any improper access to the arguments (such as a write directed to *b*) will also fault. At the time these segments appear in the address space, enough information to return is pushed on the microcode-only call stack. The process then changes to the new address space and transfers to the specified routine. This routine runs, adding the value of its second argument to its first argument, by doing

```
ADD     0(PR9),0(PR8) .
```

Next it returns, which simply has the microcode read the data pushed on the microcode stack and restore the state of the process, returning to the point after the ***Call*** instruction.

9.6. Timings

After writing and debugging the microcode and the PDP-11 assembly kernel, we measured the actual times required to perform the basic operations (*e.g.*, ***Call***, ***Call/Fork***, etc.) to see if they matched expectations. This section describes these measurements.

The biggest problem in interpreting the results had to due with scheduler overhead. The scheduler overhead is trivial, and can be ignored, in many systems. For quite a while, this was the attitude taken

with ECHOES; it led to artificially large values for all the measurements. The "system expects" the user process to run with the 60-Hertz clock interrupt enabled, and it is this interrupt that causes the scheduler to be invoked. The time that it takes to take a process off of a processor and load another process on that processor is large. No real attempt was made to make this an inexpensive operation, and it turns out the the actual cost of this is such that running with these interrupts enabled causes an effective slowdown to 60% of the raw processor speed. This is due to approximately a cost of 9900 microseconds to save a process, select another runnable process and load that process onto the processor. This time is not at all disappointing, since no attempt was made to optimize this path through the system.

In all of the figures presented here, the measurements were made with the 60-Hertz clock interrupt enabled, and the timings were multiplied by 0.6. The assumptions made are as follows. We assume that the timings in the LSI-11 handbook are correct as far as the execution time required for instructions that execute with unmapped code and operand references.

In addition, this table summarizes the times assumed for various Kmap-associated operations.

Time from memory request to context activation		2.8
This comes from the following:		
Three map-bus transactions	1.5	
Kbus and map-bus arbitration	1.3	
Further we count		
Kmap-generated memory references		5.0
Each Pmap microcycle executed		.15

Control-stack mapping operations are commonly performed while running ECHOES. They occur when the LSI-11 hardware decides to do something, such as fetching interrupt vectors, or pushing the PC and PS on the LSI-11 stack which requires the LSI-11 running the program to be in a known state (*i.e.*, the interrupt vectors have to be addressable and contain the proper data, and SP must point into an addressable region of memory.) Since this can not always be guaranteed, the Slocal marks these special memory references for the Kmap and allows it to map the references as it desires. These references are called control-stack references. The number of Pmap cycles executed in a control-stack operation is 22.

A double-length control-stack operation is one which is used to push or pop an item from the control stack. A single-length control-stack operation is one where an interrupt vector is being read. A double-length control-stack operation makes two memory references instead of one, and this takes an extra 20 microcycles in the Pmap on top of the time required to make the additional memory

reference. Thus we get the following expected times for control stack references.

Single length control-stack operation	11.1
Double length control-stack operation	19.1

I assume that interrupt latency for a NXM fault is 16.5 microseconds, that is, the same as for an I/O interrupt.

Setting the PC. On LSI-11's, it is very difficult to set the PC, or any other LSI-11 registers, from the Kmap. This is due to the fact that the PC in the LSI-11 is not accessible from the Kmap. This causes extraordinarily large times for domain switching, since the operations that switch domains, *i.e.*, ***Call***, ***Call/Fork*** or ***Return***, also change the PC.

The method used to set the PC from the Kmap is quite simple. It is also quite slow. The Kmap writes the PC to branch to in a special location in the LSI-11's memory and then forces the LSI-11 to take a non-existent memory reference fault or NXM. The first thing the NXM interrupt handler does is check for whether the reason for the NXM is to cause a PC reset and if so the NXM handler simply returns to the location where the branch is to be made.

NXM handler execution costs about 193 microseconds.

9.6.1. Procedure *Call* and *Return*

In this case, we consider how much time was spent doing a procedure call and return, in the case where no fork was requested.

The time was 846 microseconds for both operations, of which 386 can be written off to the overhead of setting the PC. This leaves 460 microseconds, the equivalent of approximately twenty-four 16-bit mapped memory references.

I will now explain what this time was used for.

The *Call* Operation. ***Call*** pushes several items onto the protected call stack (segment 5 in the address space). Specifically, the data that are pushed are the 32-bit return address, the 16-bit domain field, the number of arguments (in a 16-bit field), and the stack bound (a 32-bit quantity). This is a total of 6 memory references.

In addition, the actual instruction that causes a ***Call*** to be performed takes 3 memory references to be fetched, and another 2 to locate the address to which to branch.

The total amount of time that should be required is thus approximately 220 microseconds for a

Call. The actual measurements are described later.

The *Return* Operation. The ***Return*** operation should take two 16-bit mapped memory references less than the ***Call***, since it pops the same information back off from the the stack and restores the processor state fields from it, but doesn't fetch an address to which to branch, since it is getting the transfer address from the call stack.

Thus the estimated time for a ***Return*** operation is 180 microseconds.

The number of microcycles to handle a ***Call*** plus a ***Return*** total to about 382 microcycles. Nearly all of the operations performed during a call or a return could be done in parallel, I suspect, if the machine were designed to do *these* operations fast, but the Kmap wasn't designed to run ECHOES. The 382 microcycles amount to another 57.3 microseconds. The total cost should be 57.3 + 180 + 220 = 457 microseconds. This is very close to the observed 460 microseconds. LSI-11 speeds can vary about 6% from the listed speeds in the handbook. Thus the figures obtained by measurement are in substantial agreement with what is expected.

9.6.2. *Call/Fork* Operations and Their Return

The ***Call/Fork*** operation takes about 172 extra microseconds, when a fork is not created. The additional cost comes from several places. First of all, there is a ***Synchronize*** operation performed to wait for the fork to complete its execution.

Second, the ***Call*** operation, when told to fork, writes out the synchronization information, which is a 16-bit word naming the process that will be executing the fork, or -1 if the caller's process will execute the called procedure. In addition, instruction fetches use another six mapped 16-bit fetches. Another such fetch is made by the Kmap to read the word specifying which process is being waited upon when the wait operation is performed. This is a total of 8 memory references, all mapped. However, this is mitigated somewhat by the fact that the ***Call*** itself doesn't push the two 16-bit words that normally contain the return address. Thus there is a net gain of 6 mapped memory fetches, which should be about 120 microseconds

The difference (52 microseconds) from the observed could be due to several things, including time to execute the somewhat more complex microcode and the execution time of the ***Synchronize*** operation used to trigger the Kmap's actually doing the operations above the instruction fetch already counted.

If the fork actually succeeds, then the figures for the time to perform a ***Call/Fork*** operation are nearly identical when two Cm's are running. In particular, the cost goes from 1018 microseconds (1

Cm available, thus none available for a fork) to 1012 microseconds (2 Cm's available, 1 is available for the fork). If another processor is added, then the time to execute a ***Call/Fork*** operation *increases* to 1278 microseconds, and when yet another processor is added to the configuration, the total elapsed time increases to 1573 microseconds. The initial drop in the execution time could be due to actual parallel execution of the caller and the forked process. The further *increases* in execution time are almost certainly due to memory and Kmap contention.

9.6.3. Timing Summary

Operation	Microseconds
Standard procedure ***Call*** and ***Return***:	846
Failing ***Fork*** (user requested a new process for the ***Call***ed procedure but didn't get one), ***Return*** and re-***Synchronize***:	1018
Successful ***Fork*** (user requested a new process for the ***Call***ed procedure and got one), ***Return*** and re-***Synchronize*** processes (2-Cm system):	1012
Successful ***Fork***, ***Return*** and re-***Synchronize*** (3-Cm system):	1278
Successful ***Fork***, ***Return*** and re-***Synchronize*** (4-Cm system):	1573

9.7. Conclusions

The main conclusions that can be drawn from ECHOES are that the proposed addressing architecture can actually be implemented, resulting in a system enabling very fast calls between protected subsystems. Furthermore, these calls can be executed in parallel with the continued execution of the caller if this is what the programmer desires without greatly increasing the time required for the call and return from the protected subsystem. We are able to do this by separating out the expensive address space creation operation from the relatively cheap process creation.

10. Comparison of the Cm* Firmware Machines

Pradeep Sindhu, Steve Vegdahl, and Anita Jones

This chapter compares the performance of three of the microcode systems that have been written for the Cm* Kmaps: SMAP, MEDUSA, and STAROS. The systems are compared by giving the costs of performing equivalent, or at least similar, operations from each of the three systems, and then discussing the reasons behind any differences in performance. The operations selected for measurement are the ones that are expected to be executed relatively frequently by user programs and the operating system. These operations are therefore the ones that will have the greatest impact on overall system performance. The selected operations perform the following functions:

- Mapped memory references
- Change of addressability
- Synchronization
- Message communication.

The cost of mapped memory references is discussed first, both for the intracluster and intercluster cases. Mapped memory references were measured in an otherwise idle system and under various loads to determine the range of performance to be expected. The next operation measured, change addressability, allows a process to change a portion of its address space to make a new object accessible. How frequently this operation is executed in practice depends strongly on the application program, so its effect on overall performance is hard to estimate without additional data. It is, however, of concern since the address space of an LSI-11 is only 16 bits. A typical synchronization operation was chosen for measurement because synchronization costs are important, especially for processes that share read/write data.

The last set of measurements in this chapter are for the message communication operations of MEDUSA and STAROS. SMAP does not implement messages. Both operating systems rely on message communication where more conventional operating systems use procedure activation. In addition, messages are used for synchronization and for interprocess communication. The extensive use of messages in both systems means that the performance of message operations is important to the overall performance of both systems.

One of the problems in comparing the performance of the three systems has been that the systems are not functionally identical. In fact, SMAP is much more primitive than either MEDUSA or STAROS, and provides substantially less functionality than either of these two microcodes. SMAP was written to provide a simple, logically uniform addressing environment along with a few synchronization primitives that could be used in writing parallel programs. Both the STAROS and MEDUSA microcodes,

on the other hand, were written to support particular operating systems and consequently provide a much more powerful and convenient environment than SMAP. There are also differences between the functionality provided by nearly equivalent operations in MEDUSA and STAROS that make direct comparison somewhat difficult. These differences will be brought up as the individual operations are compared.

In order to appreciate the measurements that are presented in this chapter, something needs to be said about the degree to which the different systems have been tuned. SMAP was written several years ago and has not been optimized since it was first written. However, because the first implementation was coded fairly carefully and the system is reasonably simple, one cannot expect large improvements by making refinements. The version of the MEDUSA microcode that was used in the measurements of this chapter is the first implementation of the system. Execution efficiency was a primary goal, but critical functions have not been subsequently optimized. The initial version of the STAROS microcode, on the other hand, sacrificed execution efficiency in order to make it easier to write and maintain. Currently, optimizations for speed are being made where necessary in STAROS, and measurements in this chapter reflect some of these optimizations.

Most of the performance measurements in this section were obtained by measuring the operations directly on the hardware. The remainder were measured using the CYCLES program for tracing through source microcode (see Section 4.6). The experiments described in that section show that, for intracluster operations, measurements obtained by tracing are in close agreement with measurements made directly on the hardware. Nonetheless, CYCLES measurements are marked explicitly so that the method of measurement is clear. Although the figures for operations measured on the hardware were computed by averaging over a large number of repetitions, strictly speaking, they have not been statistically validated because all of the repetitions used the same hardware components. Variations between the speeds of interchangeable components could cause the figures to vary up to about 10% for some of the operations. For the more complex operations which use a greater variety of components, variations in the individual components may average out, leading to a smaller variance than for the simpler operations.

10.1. Mapped Memory References

The Kmap operation whose performance is perhaps the most critical to system performance is the mapped memory reference. Because of the distributed nature of the Cm* hardware, mapped references entirely within a cluster are necessarily faster than mapped references involving another cluster. Thus, two measurements are needed to characterize the performance of this operation—the time to make a local cluster reference, and the time to make an intercluster memory reference. The

table below shows the times for each of the three microcodes measured in an otherwise idle system.

Time for a single mapped memory reference:

	Intracluster	*Intercluster*
SMAP	8.3 μseconds	26.2 μseconds
MEDUSA	8.3 μseconds	30.8 μseconds
STAROS	8.6 μseconds	35.3 μseconds

Note that these timings cover the elapsed time from the moment a processor initiates a mapped read reference to the moment it receives the result and is able to continue. The above numbers were all measured with exactly the same configuration and exactly the same hardware. Thus, they are accurate relative to each other but may vary slightly in an absolute sense when a different computer module or a different cluster is used for the same experiment.

10.1.1. Intracluster References

For intracluster references, all three systems do essentially the same thing: they add an offset to the base address of a segment to compute the physical address of the word to be referenced, and then perform a Kbus operation to make the memory reference. MEDUSA and STAROS in addition perform bounds, type and rights checking to make sure that the reference will not violate protection constraints. It can be seen that the times for the three systems are quite close. The extra time for STAROS is because of two additional Kmap microcycles during cache lookup.

The cache structures of MEDUSA and STAROS reflect different tradeoffs between the speed of memory references and the cost of purging a descriptor from the cache. In MEDUSA's cache structure, if a window points to a descriptor then the descriptor is guaranteed to be valid (see page 150). Thus, a mapped reference need not check whether the descriptor pointed to is the correct one. In STAROS, on the other hand, a window may point to an incorrect descriptor, so each reference must verify the validity of the descriptor. It is this check that accounts for the extra microcycles in a STAROS mapped reference. MEDUSA avoids it by linking together all windows that point to a descriptor so that the windows can be invalidated when the descriptor is purged from the cache. This guarantees that a window is either null or points to the correct descriptor. STAROS does not require such links because a descriptor can be purged without changing the pointers in the windows that refer to it.

Figures 10-1 and 10-2 characterize the performance of intracluster memory references when several computer modules are making simultaneous references. Figure 10-1 shows the throughput when all computer modules are making references to the same memory location. Hence, as the number of modules making references increases, it is the Slocal and the memory that become

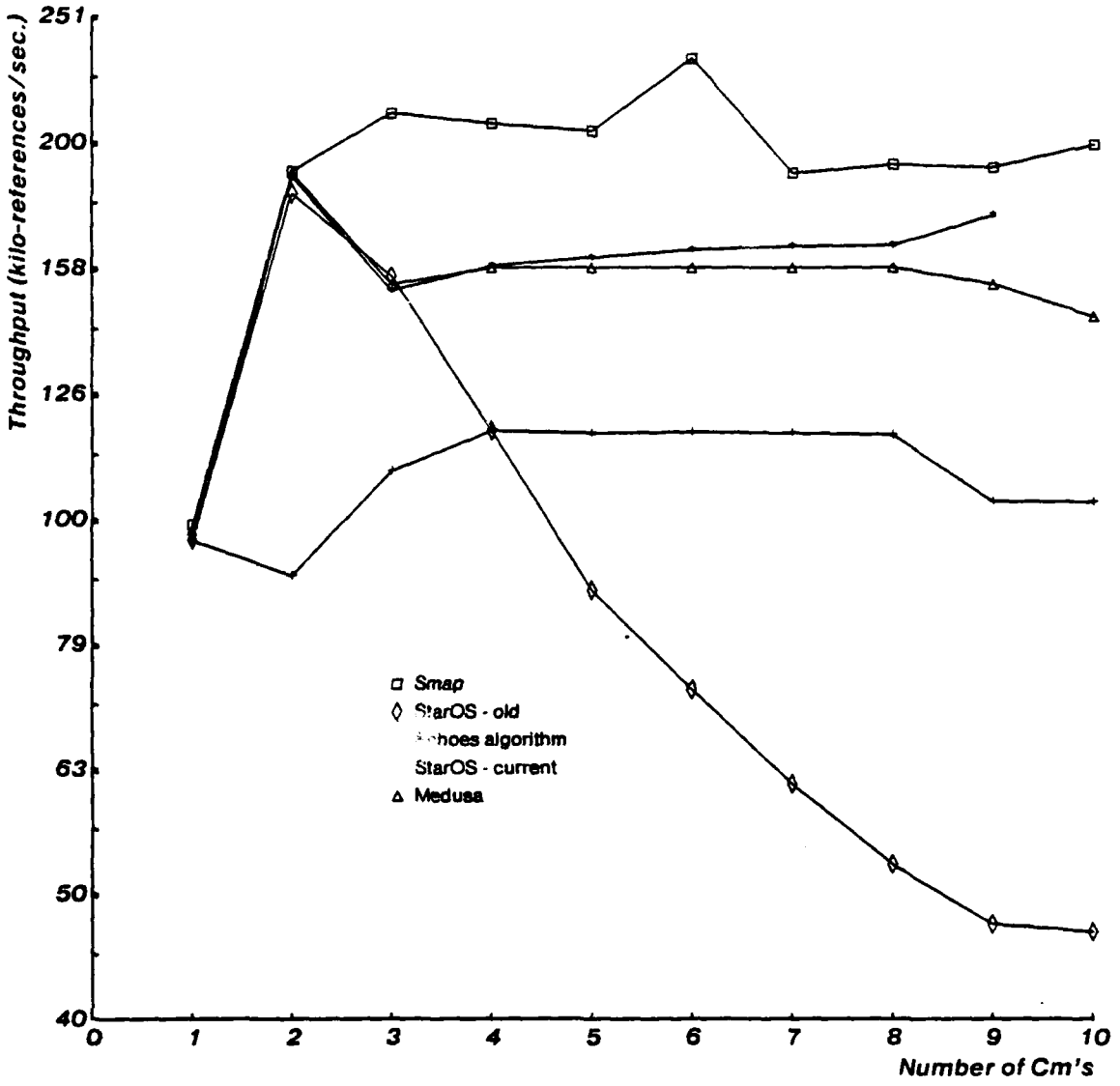


Figure 10-1: Intracluster Throughput with Slocal Contention

bottlenecks. For all microcodes the throughput increases until about two or three Cm's and then levels off. The maximum throughput supported by SMAP (210K references/second) is essentially determined by the hardware because little time is spent in the Kmap. However, the simple algorithm that SMAP uses to resolve contention at the destination Cm could cause references to be starved. The maximum throughput for MEDUSA is 189K refs./sec., and that for STAROS is 187K refs./sec. Both are

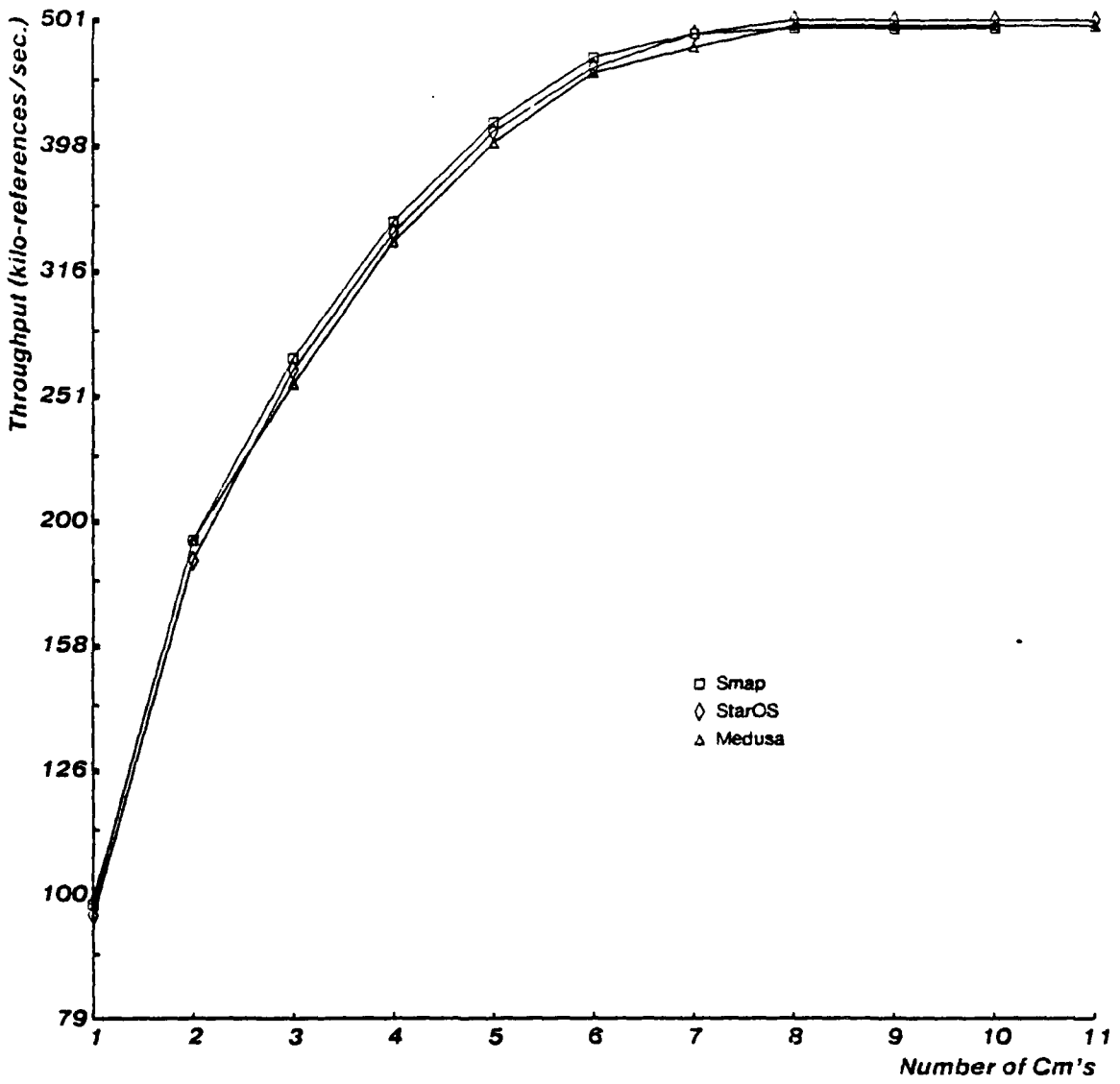


Figure 10-2: Intracluster Throughput without Slocal Contention

lower than SMAP. However, there is no possibility of starvation because requests are serviced in a manner that ensures no request will wait forever. The importance of the algorithm used to resolve contention for the Slocal is discussed further in Section 10.1.3.

Figure 10-2 shows the throughput of intracluster memory references in the three systems when

each computer module is making references to a different Cm's memory. Each data point was measured with n Cm's in a ring configuration; that is, Cm i directed all its memory references to the memory of Cm $(i + 1) \bmod n$. In this case there is no contention for the target Cm, and the three systems perform almost identically.

10.1.2. Intercluster References

In the case of intercluster memory references MEDUSA and STAROS have to do a little more work than SMAP at the destination cluster. SMAP sends a physical address to the destination Kmap which then makes the reference; both STAROS and MEDUSA send descriptor names instead of physical addresses. MEDUSA then searches the cache at the destination Kmap to learn the physical address of the location to be referenced. STAROS sends the destination Kmap extra bits in its linc message that constitute a guess of where the descriptor is in the cache. Though not guaranteed to be accurate, the guess speeds up repetitive references as long as the desired descriptor is still present in the cache.

The optimization of transmitting a guess of descriptor location improved the intercluster memory reference time of STAROS from 41.7 μseconds in the initial, unoptimized version to 35.3 μseconds in the current version. Further improvements are expected by replacing some procedure calls by special-case in-line code to handle intercluster memory references. The current version of MEDUSA does not use the descriptor-guess strategy; this optimization could improve the intercluster reference time for MEDUSA also.

Figures 10-3 and 10-4 show the throughput of intercluster memory references for each of the systems under loaded conditions. In Figure 10-3 all references are made to the same destination cluster, but the references are distributed between the Cm's so that no Cm saturates; the destination Pmap is therefore the bottleneck. At saturation, SMAP delivers about 210K refs./sec., MEDUSA delivers about 160K refs./sec., and STAROS about 145K refs./sec.

Figure 10-4 depicts the performance of intercluster memory references when the source cluster is the bottleneck—each Cm in the source cluster makes references to a different Cm in some other cluster. The maximum throughput for SMAP is approximately 250K refs./sec., for STAROS about 150K refs./sec., and for MEDUSA about 170K refs./sec. SMAP performs substantially better in the presence of contention; however, the algorithm it uses may cause starvation. The decrease in throughput between 7 and 8 Cm's for MEDUSA and a much smaller decrease for SMAP between 8 and 9 Cm's are both due to their scheme for avoiding deadlock over contexts. MEDUSA's scheme is more costly, but it guarantees freedom from starvation as well.[25] There is no corresponding decrease for the STAROS

[25] This expense could be reduced significantly if the implementation were optimized for simple operations, such as memory references.

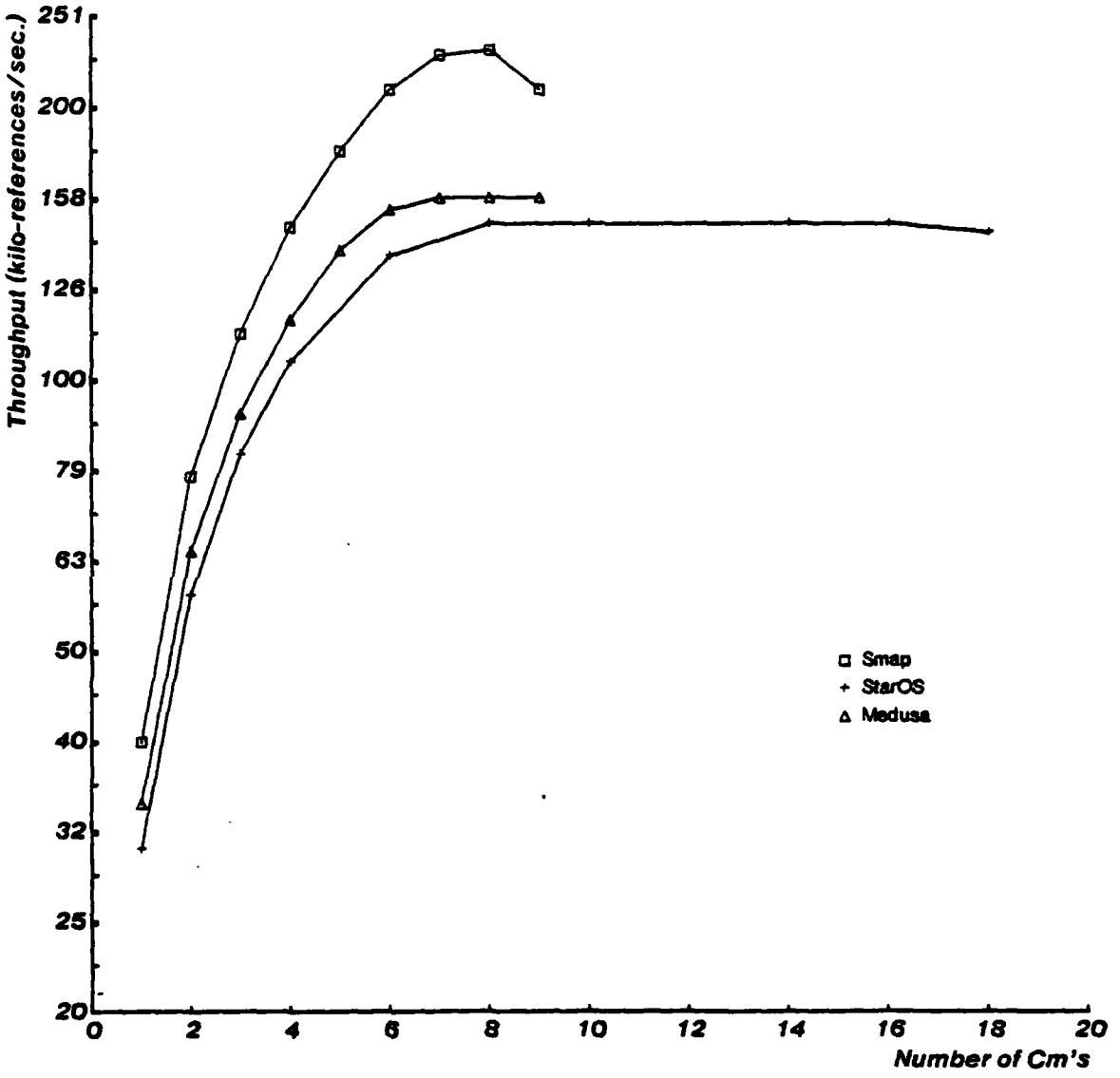


Figure 10-3: Intercluster References—Saturation of Destination Cluster

microcode because it currently does not address the problem of deadlock over contexts.

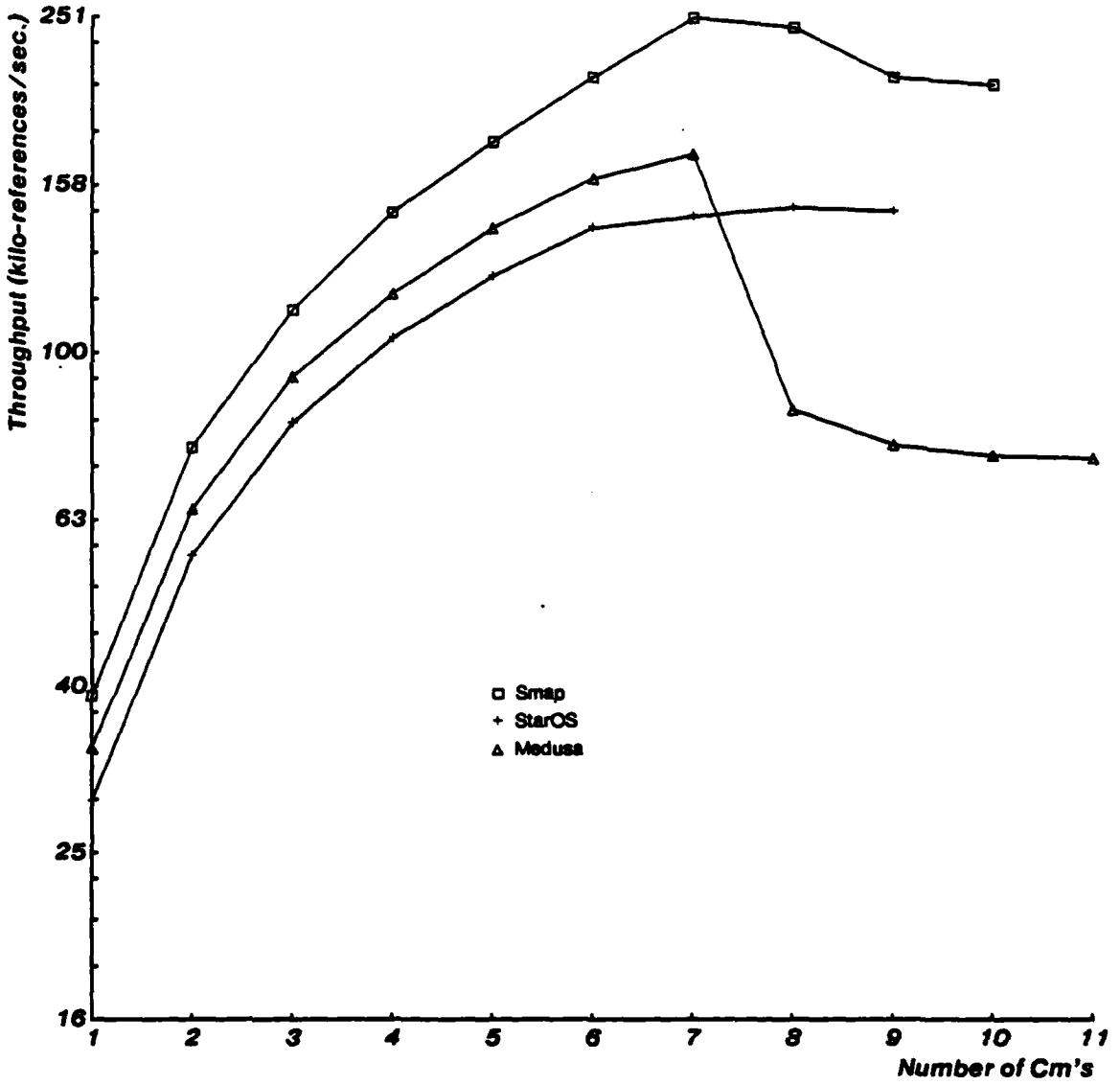


Figure 10-4: Intercluster References—Saturation of Source Cluster

10.1.3. The Importance of Contention-Resolution Algorithms

In the previous two sections we have seen the performance of mapped memory references for various conditions of loading. In this section we consider a particular contention situation and illustrate the effect of the low-level algorithm used to resolve contention on the speed of the operation

that is being performed. The point that we emphasize is that low-level contention resolution algorithms need to be designed and tuned fairly carefully since the performance of the system depends quite strongly on these algorithms even under moderate loads.

Contention occurs when multiple Cm's in a cluster repeatedly reference the memory of another Cm in the cluster. The resource contended for is the destination Slocal, and the algorithm used to resolve this contention is implemented within the Kmap that presides over the cluster. Figure 10-1 shows the performance of mapped memory references for five different implementations of a strategy for contention resolution.

The curve labelled SMAP shows the performance of the relatively simple, but starvation-prone algorithm used by SMAP. In this algorithm whenever a context servicing a request encounters a busy Slocal, the Pmap waits for 20 microcycles and then lets the context retry the reference. The wait is performed in order to make it more likely that the reference in progress completes before the retry is done. There is a tradeoff in this scheme between the length of time that the Pmap is allowed to idle and the probability that a request may have to retry many times before succeeding. Increasing the wait time lowers the probability of starvation but wastes Pmap cycles, thereby slowing down requests for other operations that could proceed in parallel.

The curve marked STAROS-old illustrates the performance of the algorithm formerly used by STAROS. This algorithm is starvation free, but the number of Pmap cycles it expends per completed memory reference increases roughly linearly with the number of contexts that are contending for the Slocal. The effect of this dependence is that beyond about three Cm's, the Pmap becomes the bottleneck and the throughput actually decreases as additional Cm's make references. There is thus a marked difference in throughput between a constant-cost strategy and a linearly increasing cost strategy for contention resolution.

The three remaining algorithms all incur constant cost and are starvation free. They implement slightly different schemes for recording the contexts waiting to complete a reference and slightly different decision procedures for selecting the waiting context which proceeds when the Slocal becomes available. The curve marked STAROS-new illustrates the algorithm that is currently used by the STAROS microcode. A bit vector is used to record contexts that are waiting for the Slocal. The strategy used for selecting the context that will next access the Slocal guarantees that starvation does not occur. This scheme uses between 20 and 24 Pmap cycles per completed memory reference, and is the least expensive of the last three algorithms. An interesting aspect of the implementation is that the number of Pmap cycles per completed reference actually decreases as the number of contexts increases. This decrease is easily noticeable as a gradual increase in throughput as the number of Cm's is increased.

The algorithm used by MEDUSA involves placing waiting contexts in a FIFO queue.[26] The cost of insertion and deletion is independent of queue length. This scheme uses between 27 and 31 Pmap cycles per completed memory reference and is therefore slightly slower than the bit-vector implementation of STAROS under heavy contention.

The last algorithm we characterize is the one used by the ECHOES microcode. The ECHOES implementation also uses a FIFO queue to record contexts waiting for a busy Slocal. However, since the implementation was coded in MUMBLE and since no special effort was made to optimize it, the throughput is not as high as it is for the other systems—between 46 and 52 cycles are expended per completed reference.

This comparison shows that both the implementation of a contention-resolution algorithm and the algorithm itself can have a substantial influence on the speed of mapped memory references. The key difference between the different algorithms and the different implementations is the degree to which they cause the Pmap to become a bottleneck in the process of resolving contention—the implementation with the smallest Pmap cost has the greatest throughput.

10.2. Change Addressability

The change-addressability operation redefines the binding between a window and the physical memory to which the window refers. SMAP's version of the operation is the simplest and also the fastest. It associates an arbitrary physical address with a specified window by loading the window register in the Kmap. However, in order to complete change of addressability the LSI-11 has to execute one more instruction to load the Slocal register corresponding to the window (STAROS and MEDUSA do this load automatically as part of the change-addressability operation). In order to make the operations in the three systems more comparable, the 33 μseconds quoted in the table below for SMAP include the time to execute this extra instruction.

Time for change of addressability:

SMAP	33 μseconds	
MEDUSA	69 μseconds	
STAROS	109 μseconds	(traced using CYCLES)

The costs listed above for both STAROS and MEDUSA are incurred on two separate occasions. The first portion of the cost is incurred when the operation is invoked by the LSI-11. At this time the name

[26] The Kmap hardware prevents requests from being strictly FIFO, but this scheme comes as close to FIFO as possible given the hardware.

of the new object is bound to the window by loading the Kmap's window register and writing the name out to the process state kept in main memory. The second portion of the cost is incurred when the first memory reference is made through the window just loaded, in part due to the need to write the Slocal register. The extra time required by StarOS over Medusa is partly because of the three-level address structure of StarOS—two extra references to translate the C-list index into a capability—and partly because the entities that are read or written to memory during the operation are larger for StarOS than for Medusa—StarOS uses two-word capabilities, while Medusa uses one-word descriptor indexes.

10.3. Synchronization Operations

All three microcode systems provide indivisible operations that allow parallel programs to synchronize their operation on shared read/write data. The representative operation, ***Indivisible Increment***, has virtually the same semantics in all three systems. The operation costs are for the case in which the target of the increment is in the same cluster as the invoker of the operation. As is the case for the other operations in this chapter, any address information needed to perform the operation is assumed to be present in the Kmap's cache:

Time for ***Indivisible Increment***:

Smap (*immediate address parameter*)	25 μseconds	
Medusa (*immediate address parameter*)	33 μseconds	
Medusa (*virtual address parameter*)	47 μseconds	
StarOS (*virtual address parameter*)	95 μseconds	(*traced using Cycles*)

Smap's ***Indivisible Increment*** is faster than that of Medusa or StarOS because Smap makes no main-memory references to fetch the parameter for the operation; the parameter is a 16-bit immediate address that is passed up to the Kmap when the operation is invoked. Medusa has two forms of the operation; one uses an immediate address and the other a virtual address. The 14 μseconds cost difference is due to the two additional main-memory references to read the larger virtual address parameter. StarOS does not currently have an immediate address version, so its operation also pays for the memory references to read the parameter. StarOS's operation takes 48 μseconds longer than Medusa's because—

- StarOS has a three-level address structure. Two additional references must be made during each operation to read the capability.
- StarOS uses more high-level microprocedures during the operation than Medusa. In particular, the parameter-block binding to the window is recomputed for every word that is read. StarOS permits a parameter block to be spread across two windows; Medusa does not.

10.4. Message Operations

The message system is one of the most important facilities provided by the MEDUSA and STAROS microcodes. Messages are used for general-purpose communication and synchronization of processes. In addition, both operating systems are structured so that asynchronously executing "servers" provide a substantial portion of the system functions. A request to such a server is in the form of a message that conveys the requisite parameters. Likewise, any reply is transmitted as a message. Not only do both operating systems offer a message-communication facility to user programs, they also make extensive use of it internally. Performance of the message facility is therefore crucial to the efficient operation of both systems.

Message communication is the least similar of the facilities provided by the microcodes of MEDUSA and STAROS. The two differ both in function as well as in implementation. The message mechanisms have been described in earlier chapters, but it will be helpful to review them.

Both systems support buffering of the messages which are transmitted between processes. Both provide two kinds of operations: conditional and unconditional. Conditional operations run to completion regardless of the state of the mailbox or pipe that is the target for the operation. For example, when a process performs a ***Conditional Receive***, a message is returned from the mailbox if the mailbox is nonempty and a status indicating an empty mailbox is returned if the mailbox is empty. In either case the process continues processing after performing the operation. The semantics of unconditional operations are rather different in the two systems and will be discussed below. With this rough background we itemize the difference between the two mechanisms—in particular, we cite differences relevant to the performance of the message operations.

- The difference between the unconditional operations provided by the two systems is that when an operation cannot be performed immediately, the invoker of the operation is blocked automatically in MEDUSA whereas the invoker is *not* blocked automatically in STAROS. Blocking is a separate operation that is invoked explicitly by a process in STAROS.

- In MEDUSA when an activity is blocked as a result of a message operation, the activity does not relinquish its processor immediately. Instead, it retains its processor for a period called the *pause time* (see Section 8.1.2.1) that is specifiable by the activity. If the activity becomes runnable during the pause time, it is able to resume processing immediately without incurring the context-swap time of the LSI-11.

- Because a STAROS process may have several unsatisfied, unconditional requests to receive a message outstanding, delivery of several messages may be overlapped. In MEDUSA, messages arrive in the receiver's address space sequentially and while the activity is actively performing the ***Receive*** operation, or suspended awaiting the time at which the ***Receive*** in progress can complete.

- The semantics of the ***Send*** and ***Receive*** operations in MEDUSA are symmetric. Thus, for example, the effect of doing an ***Unconditional Receive*** on an empty pipe is the same as the effect of doing an ***Unconditional Send*** on a full pipe—the invoker is blocked. In STAROS, ***Send*** and ***Receive*** are not symmetric since the ***Unconditional Receive*** does not have a counterpart. If a ***Send*** is done to a full mailbox, the invoker is informed that the operation cannot complete immediately.

- The two mechanisms differ in the size and the type of entities that may be transported in messages. STAROS transmits either a one-word data message or a single capability which is the name of an arbitrary object. MEDUSA transmits variable-size data messages ranging from 0 to 4000 bytes by copying the message;

- The delivery of a message to an activity that has performed an ***Unconditional Receive*** on an empty pipe is handled entirely within Kmap microcode in MEDUSA. The corresponding delivery of a message to a process that has done a ***Registered Receive*** in STAROS is currently implemented in operating system software.

- All data transfers performed during the message operations of MEDUSA are handled by a block-transfer mechanism. Block transfers are not used in the STAROS message mechanism since blocks of data are transferred by passing a capability.

- Both facilities have an upper limit on the buffering capacity of a mailbox or pipe. The buffering capacity of a STAROS mailbox may range from 1 to 2044 data messages or 1 to 252 capabilities. The buffering capacity of MEDUSA pipes is a function of message size; it ranges from 1000 messages for zero-byte messages to a single message if the message is larger than 2000 bytes.

The differences in the two message facilities are sufficient that the operations are not directly comparable because they are not functionally equivalent. Moreover, because they are different, programmers will use them differently. The next sections give performance measurements of the two mechanisms. We endeavor to point out where the differences substantially affect the way in which the two facilities are expected to be used.

10.4.1. Performance in the Non-Waiting Case

This section discusses performance of the message operations in the situation that the operation can be completed by simply manipulating the mailbox or pipe; *i.e.*, the invoker need not wait for some action by another process or activity. The operations chosen for measurement are ***Conditional Send*** and ***Conditional Receive***, which are provided by both microcodes.

Timing measurements were taken of a single process or activity which first performed a ***Conditional Send*** of a message to an empty pipe or mailbox. As a result the message is buffered. The process or activity then performs a ***Conditional Receive*** from the same pipe or mailbox. The ***Conditional Receive*** removes the message from the mailbox or pipe, leaving it empty, and then returns the message to the invoker. Thus the message is copied twice. For STAROS, the operation

was measured using both a capability message and a data message. For MEDUSA, the message consisted of a single data word.

Measurements were made using two different distributions of the various entities involved in the operations: in the *local cluster* case the sender, the message and the mailbox are all kept in the same cluster; in the *non-local cluster* case the sender and message are in one cluster whereas the mailbox is in a different cluster. The tables below show the times for ***Conditional Send*** and ***Conditional Receive*** that were computed from from the total send + receive time obtained by measurement. The cost of the send + receive sequence was assumed to be distributed between the ***Conditional Send*** and ***Conditional Receive*** in the same ratio as the costs of these operations that were computed from CYCLES traces.

Cost for ***Conditional Send***:

	Local Cluster	*Non-Local Cluster*
MEDUSA	336 μseconds	339 μseconds
STAROS (*data*)	110 μseconds	146 μseconds
STAROS (*capability*)	151 μseconds	196 μseconds

Cost for ***Conditional Receive***:

	Local Cluster	*Non-Local Cluster*
MEDUSA	341 μseconds	342 μseconds
STAROS (*data*)	118 μseconds	156 μseconds
STAROS (*capability*)	180 μseconds	225 μseconds

The time to perform the operation in MEDUSA is roughly three times slower for the local-cluster case and about two times slower for the non-local cluster case. There are three reasons for this difference in performance. First, the block-transfer mechanism adds considerable overhead to the cost of moving small messages. Second, a deadlock-and-starvation algorithm is executed to ensure that the three contexts allocated during each operation are acquired in a safe fashion. Finally, some cost is incurred because code to handle waiting is executed, even though it is not necessary in this particular case.

10.4.2. Performance in the Waiting Case

This section gives the performance of the message mechanisms under the assumption that an activity or process has to wait for some other process or activity to act. In particular, we consider a two-process message interaction that involves sending a message from one process to another process that is waiting after having done an ***Unconditional Receive*** on an empty mailbox or pipe. The performance of the process-process interaction under the assumption that the receiver is *not* waiting can be obtained from the measurements of the previous section.

There are two time measurements of interest. One is the process-to-process interaction time, that is, the elapsed time from the instant that the ***Send*** operation is invoked to the instant the waiting receiver is about to execute its first instruction following message delivery and any context swap that may be necessary. The second time is the duration of the operations, that is, the processor time that is expended for that operation.

First, we consider the process-to-process interaction time. There are two cases for MEDUSA and one for STAROS. Recall that in MEDUSA an activity that does a ***Receive*** on an empty pipe is not swapped off the processor until its pause time is exhausted. The first case corresponds to the arrival of the message before the the pause time is exhausted; in this case no context-swap overhead is incurred and the complete interaction takes 484 μseconds. In the second case the message arrives after the pause time has been exhausted, so a context swap is incurred. In STAROS the ***Registered Receive*** is a separate operation from ***Block***, an operation that renders the process non-runnable and initiates a context swap. If the message arrives after the ***Registered Receive*** and before the process ***Block***s, then no context swap occurs.

Several assumptions were made in deriving the numbers that have been marked as estimates. First, the activity or process is assumed to have been waiting long enough so that a switch to another process is completed before the ***Send*** is done. Second, the context-swap times included in the timings are assumed to exclude the time to make any policy decisions about which process is to execute next—only the time to make the actual process switch is included. Finally, in the case of STAROS we assume that ***Block*** immediately follows the ***Registered Receive*** and no message could have arrived in that interval.

Process-to-process interaction times:

MEDUSA (*within pause time*)	484 μseconds	
MEDUSA (*pause time exhausted*)	1600 μseconds	(*estimated*)
STAROS	4000 μseconds	(*estimated*)

The reason that STAROS incurs a greater cost than MEDUSA is because both context swap and delivery of a message into the address space of the receiver are currently implemented in software. The 4-msec. cost is about equally divided between them. Replacing the software implementation with microcode would substantially reduce the cost. And if message delivery were in microcode, it would be possible to implement a version of the pause strategy in STAROS for experimentation. A process would be made to pause by executing code which expanded as part of the BLISS macro used to invoke the STAROS ***Block*** operation. The embedded code would consist of a variable length, locally executed, busy-wait loop.

The second cost of interest is the duration of the ***Send*** and ***Receive*** operations in the waiting

case. The mailbox or pipe is empty and a receiver unconditionally requests a message. The message is a single data word. For StarOS the times are constant. Both Medusa operation times are a function of the size of the message. The duration of ***Unconditional Receive*** also depends upon how much of the pause time elapses before the message arrives.

Duration of ***Send*** and ***Receive*** operations in the waiting case:

Medusa ***Send***	490 μseconds	(*estimated*)
Medusa ***Receive***	250 μseconds +	elapsed pause time (*estimated*)
StarOS ***Send***	2000 μseconds	(*estimated*)
StarOS ***Receive***	166 μseconds	(*traced using* Cycles)

The StarOS ***Send*** time is dominated by the cost of software implementation of message delivery. It should be noted that during a Medusa message operation, both the sender's and receiver's processors enter a pause state during which they may process interrupts. The sender's processor is in pause state during the block transfer; the receiver's processor is in pause state for the duration of the pause interval.

10.4.3. The Tradeoff Between Passing Pointers and Copying Data

The message mechanisms of Medusa and StarOS use different approaches for transferring messages from a sender to a receiver. StarOS can pass global names in the form of capabilities; hence a process or a mailbox can be passed as a message. Medusa messages are in the form of variable-size blocks of data; hence one activity cannot dynamically send a pipe to another activity.

If we confine our attention to messages that are in the form of blocks of data, we can analyze the tradeoff between passing pointers and copying data. The cost of accessing non-local data in Cm* is significantly higher than the cost of accessing local data. So it is not immediately clear at what point it becomes cheaper to move the data itself instead of accessing it non-locally. The purpose of this section is to quantify the nature of this tradeoff, and to indicate the range of values for which one scheme or the other is more efficient.

There are two parameters that determine which mechanism is more efficient. The parameter that has the greater effect is the number of times each datum will be referenced by the receiver. We will call this parameter *frequency of use*, and will examine three regions: *sparse use*, where only a small fraction of the words transported are actually referenced; *moderate use*, where each word is referenced exactly once; and *heavy use*, where each word is used a number of times by the receiver. The second parameter is the number of words that need to be transported; we will consider two cases: the time to communicate one word, and the time to communicate 2048 words.

In each case, the cost will include the time for actually transferring the message as well as the time that receiver must spend in order to access the words transported. Thus the StarOS measurments include the time to make the words addressable via the capability used in the transfer.

Sparse Use. Let us assume that only a very small fraction of the words transported are ever referenced. This would be the case, for example, if the data transferred was sorted and the receiver was doing a binary search to locate a particular value. The times for 1 word and 2048 words are given below. There is no break-even point since the pointer-passing scheme is uniformly better.

	1 word	*2048 words*
Medusa	484 μseconds	21000 μseconds
StarOS	228 μseconds	440 μseconds

Moderate Use. Let us assume that each word transported is referenced exactly once. Under this usage pattern, the factor that determines which scheme is better is the cost of the non-local access. For the local-cluster case it is uniformly better to pass a pointer, although the cost difference is small. In the non-local cluster case the choice is influenced by whether the message is buffered in the Medusa or not. If data is not buffered, it is better to move the data as long as more than 8 words are being transported: If data is buffered, then it is better to move the data only if it is greater than 1345 words.

One frequent use of messages is to communicate the parameters for a function invocation. Such messages are expected to be of modest size and to receive moderate use. It is cheaper to send such a message by pointer except when the message is buffered and transported to a remote cluster.

	1 word	*2048 words*
Medusa (*no buffering*)	484 μseconds	27000 μseconds
Medusa (*buffering*)	677 μseconds	48000 μseconds
StarOS (*local cluster*)	228 μseconds	21000 μseconds
StarOS (*non-local cluster*)	320 μseconds	73000 μseconds

Heavy Use. Let us assume that each word transported is referenced a large number of times, so that it always pays to move the words. In the table below, we list the costs incurred by the two systems. For StarOS, we assume that the block transfer is coded in software, using the most efficient implementation permitted by the LSI-11 processor. The break-even points for the intra- and intercluster cases are 28 words and 5 words. That is, if more than 28 words need to be transferred intracluster or more than 5 words intercluster. then it is cheaper to move them in Medusa than in StarOS.

	1 word	*2048 words*
Medusa	484 μseconds	27000 μseconds
StarOS (*local cluster*)	228 μseconds	45000 μseconds
StarOS (*non-local cluster*)	320 μseconds	90000 μseconds

ace Costs

1 and 10-2 show the number of microinstructions used by MEDUSA and STAROS for rious operations. It is difficult to compare these numbers, because the two microcodes architecture and structure, and because different methods were used in classifying the For example, *Indivisible Operations* are included in STAROS's *Miscellaneous User-Level* while *Intercluster Communication* is not distinguished in the MEDUSA classification are spread over the various parts of the system.

Table 10-1: The Sizes of Various Portions of the MEDUSA Microcode

nction	Number of Microinstructions
essages and events	901
ommon routines	750
obustness	670
ddressing structure	610
lock transfers	367
terrupt handling	285
ctivity multiplexing	157
ndivisible operations	105
otal	3845

e seen from the two tables that the MEDUSA microcode uses more space than STAROS. One sons for this is that tradeoffs in MEDUSA were generally made in favor of speed at the of microcode space. The second reason is that the MEDUSA microcode has also ed more functionality in the following ways:

essage operations are more complex (see Section 10.4).
rforms memory reference retries on hardware errors, such as parity errors.
s a more sophisticated exception-reporting mechanism.
ROS has not yet implemented a *block transfer* mechanism, whereas MEDUSA has.
MEDUSA microcode provides substantially more support for activity multiplexing than STAROS microcode.

OS, the approach was taken that a simpler, more modular system could be implemented onal speed improvement and functionality could be added once the system was being used necks were identified. The STAROS group chose to restrict functions implemented in in order to reserve space for both application experiments and performance monitoring.

Table 10-2: The Sizes of Various Portions of the StarOS Microcode

Function	Number of Microinstructions
Addressing structure	857
Capability manipulation (194)	
Cached-descriptor maintenance, searching (153)	
Deque/stack references, pointer manipulation (153)	
Address mapping (126)	
User-level capability operations (118)	
Cached-window maintenance (113)	
Message operations	326
Low-level operations and initialization	317
Miscellaneous user-level operations	310
Intercluster communication	241
Garbage collection support	71
Total	**2122**

This would be more difficult in Medusa because it has used nearly all of the control store.

10.6. Interpreting the Results

This chapter contrasts the performance of three microcodes that are reasonably different from one another. Although many of the performance figures appear to be directly comparable, some care is needed in extrapolating from these measurements to predict the performance of an application running with any one of the microcodes. The reader should be aware of some of the considerations involved in such extrapolation.

Smap, Medusa and StarOS, in that order, provide increasing functional power in the addressing and protection that they provide to the software developer. For the most part, the cost of comparable functions increases in the same order. An Smap operation is generally cheaper than a comparable Medusa operation, which in turn is generally cheaper than a comparable StarOS operation. The cost difference in the execution of one invocation of an operation may be minute, or it may be orders of magnitude. Some of these cost differences are due to implementation issues as discussed before. But some cost derives from differences in service that the systems deliver.

The design of an operating system reflects a number of tradeoffs between functional power and cost. Though the operations of one system are faster, that does not mean that programs written using that system will run faster. Functionality not provided in the system may have to be implemented by

the user. Indeed, it may have to be re-implemented by each user. In the case of Cm* systems, functionality not implemented in microcode may have to be implemented in the much slower medium of software.

An operating system designer chooses what he believes to be suitable functional power delivered at an acceptable cost. The desired result is a net savings in eventual, overall system usage. The three microcodes reflect different design decisions, even different attitudes about software development. For example, although it delivers better performance as measured in mapped references per second, SMAP addressing is inadequate for incorporation in an operating system, because any executing program may change addressability so that any word in the machine can be written. Both MEDUSA and STAROS provide additional protection, and they pay for it in dynamically expended time and space.

MEDUSA and STAROS have fundamental differences. To cite one example, names in MEDUSA are always interpreted relative to an activity. In contrast, the STAROS capability is a global name. One result is that in MEDUSA communication is virtually always "by copy" where in STAROS communication can be either "by name" or "by copy". It is difficult to evaluate such differences between MEDUSA and STAROS for a number of different reasons. First, the cost may not be apparent by considering a single operation; it may be spread across the system. The incurred cost may depend on the frequency of use of selected operations. So, a single difference in performance cannot be cited. Second, differences in functionality will cause users to design and program their applications differently in the two systems. Because Cm* is a multiprocessor, the task decomposition itself may be influenced. In addition, the user will adapt his implementation to improve performance in the context of the operating system being used. So the implementation of an application on the two systems may look quite different. Certainly, the number of Cm* cycles required to compute an answer in two implementations of the same algorithm can be compared. It will be more difficult to compare programmer productivity, robustness, and ease of use.

There also exist costs incurred in invoking operations that are not reflected in the performance figures of this section. An example of this cost is the setting up of a *parameter block* to perform the operation. Because the LSI-11 is considerably slower than the Kmap, setting up the parameter block can sometimes be as expensive as the operation itself, especially if the operation can be performed rapidly by the Kmap. A discussion of the parameter-block costs in STAROS and a comparison to an operation cost is presented on page 129.

Cm* is an experimental system. And its operating system microcodes are moving targets. We have shown how optimizations can substantially alter the cost of an operation. We expect both systems will continue to evolve. Indeed, they will probably be adapted to serve particular

experimentation. In each system, choices were made about which medium—microcode or software—to use for the implementation of each function. Almost all of the message system of MEDUSA is implemented in microcode whereas approximately a third to half of STAROS's message system is in software. Performance would be noticeably influenced by changing the medium of implementation of a given function.

Yet another consideration in gauging the effect of the performance figures on overall system performance is that most of them have been measured under no load. The exception is mapped memory references which have been measured under both artificial load and under the applications discussed in earlier chapters. Some measurements of additional operations under loaded conditions are discussed in Section 8.1.6.2. More importantly, we do not yet know the mix of operations and the frequency of their invocation during application execution.

10.7. Summary

Having compared the performance and size of three of the microcode systems for Cm*, it seems appropriate to see if any conclusions can be drawn from the comparisons, and if there are any important trends that can be observed from implementing three fairly different systems.

The first observation is that a substantial amount functionality can be provided by the microcode at a reasonable cost in terms of programming time. This functionality includes address mapping, synchronization primitives, message communication, protection, and the implementation of abstract data types. Most of these functions except address mapping are traditionally implemented in operating system software, where the cost of invoking the operation itself can be quite high, often dominating the actual cost of the operation itself. Thus, there are large gains to be had in terms of operating system performance by placing commonly used functions in microcode and relatively little to be lost in terms of the time taken to implement the entire system.

An important observation that can be made from the performance of the message systems is that on Cm*, the cost of invoking functions using messages can be made comparable to the cost of invoking a high-level language procedure. Since message communication is equivalent in semantic power to the procedure call, the implication is that the cost of distributing components of the operating system or user programs is negligible as long as most of the memory references made by each component are local. Thus all of the benefits of distribution can be derived without incurring high communication overheads.

One of the lessons that has been learned from implementing a number of different microcode systems is that careful attention needs to be paid to low-level design of the microcode to ensure that

the system will perform adequately under loaded conditions. Two of the problems that arise frequently under these conditions are deadlock over shared resources and starvation of requests that are not served in a fair manner. The algorithms and system structures used to guarantee freedom from these problems may sometimes extract a heavy penalty if they are not designed with efficiency in mind, and the overall performance of the system may suffer drastically as a result. An example of this phenomenon has been described in Section 10.1.3.

Another observation is that simple operations like memory references can be performed efficiently. Intelligent use of caching and of special-case microcode to handle memory references seems to make the cost of such references fairly independent of the complexity of the address structure. However, it is important to note that a complex address structure does extract a price for the more complicated operations since it is not feasible to write special-case microcode for all such operations.

The writing of special-case microcode to improve performance is an example of a space-time-complexity tradeoff that was encountered frequently in the writing of all three systems. If a particular operation is to be optimized for speed then it often helps to write microcode that is specially adapted to the needs of that operation. However, special-case microcode not only takes up more space, but it makes the code less structured and therefore harder to modify and debug, or to adapt for the purposes of experimentation.

11. Language Support for Parallelism

Thomas Rodeheffer and Peter Hibbard

As it becomes more and more economical to build machines which can perform more than one operation at the same time the problem of arranging a program to take advantage of the parallelism which is available in such a machine is becoming more and more of interest. We are working on a compiler and run-time system for an asynchronous multiprocessor which will detect and exploit low-level parallelism inherent in a user's program.

One dimension along which the various parallelism providing machine architectures can be measured is the time required for two processing elements to synchronize with each other, relative to the time required to perform an ordinary arithmetic operation. For synchronous architectures which contain a central master clock, an array machine or a pipeline machine for example, this measure is zero, since synchronization happens implicitly with every clock tick. For a distributed system this measure may be quite large.

Previous work on the automatic extraction of parallelism during compilation (for example [Kuck *et al.* 72, Banerjee 79]) has generally assumed a machine architecture for which the synchronization time is zero. Naturally, if one can predict during compilation the exact behavior of the processing elements during execution, one has the possibility of achieving very intricate choreography at a very low level. If exact predictions are not possible, however, one must arrange for the processing elements to act in a manner less and less tightly cooperative, depending on the relative cost of synchronization and the accuracy of the predictions.

Interestingly enough, most work on language features and programming methods which allow the programmer to specify and control parallelism (for example [Ichbiah *et al.* 79a, Ichbiah *et al.* 79b]) has assumed a machine architecture in which the processing elements are largely independent and separately programmable, and in which an interprocess synchronization (or communication) might cost about the same as any other statement in the language. Work on distributed systems emphasizes machine architectures in which the synchronization costs are much higher.

We are interested in asynchronous multiprocessors in which the cost of synchronization runs from very small, say a few machine instructions, up to moderate, say a few hundred machine instructions. We feel that with the cost of synchronization in this range there remains in ordinary programs parallelism that would be worthwhile to take advantage of but whose explicit specification would be too tedious. We propose to detect and exploit this parallelism automatically.

In order to avoid other problematic aspects of multiprocessors, we assume a very simple asynchronous multiprocessor architecture: one in which each of the component processors is an identical, standard uniprocessor, one in which all memory is shared with no access interference, one in which an unlimited number of processors is available, and one in which there exists some synchronization mechanism of a particular, fixed cost by which means idle processors can be obtained when needed and active processors can communicate among themselves. Of course this architecture is not very realistic, but we view the task of the run-time system as being to provide as effective a simulation of it as possible.

What the compiler shall be concerned with is discovering parallelism in the user's program, estimating the synchronization costs required, and deciding on a plausible structure of parallel tasks for an idealized multiprocessor of given fundamental cost of synchronization.

We intend to create a compiler and run-time system which will compile programs for and execute programs on Cm*. Currently the C language is under consideration as a source language.

11.1. References

[Banerjee 79] U. Banerjee, S. C. Chen, D. J. Kuck, and R. A. Towle.
Time and parallel processor bounds for Fortran-like loops.
IEEE Transactions on Computers C-28(9):660 – 70, September, 1979.

[Ichbiah *et al.* 79a]
J. D. Ichbiah, J. C. Heliard, O. Roubine, J. G. P. Barnes, B. Krieg-Brueckner, and B. A. Wichmann.
Preliminary Ada reference manual.
Association for Computing Machinery, 1979.
published in *SIGPLAN Notices* 14(6), part A.

[Ichbiah *et al.* 79b]
J. D. Ichbiah *et al.*.
Rationale for the design of the Ada programming language.
Association for Computing Machinery, 1979.
published in *SIGPLAN Notices* 14(6), part B.

[Kuck *et al.* 72] D. J. Kuck, Y. Muraoka, and S. C. Chen.
On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup.
IEEE Transactions on Computers C-21(12):1293 – 1310, December, 1972.

Appendix I
Major Cm* References

[1] G. M. Baudet.
The design and analysis of algorithms for asynchronous multiprocessors.
PhD thesis, Carnegie-Mellon University, April, 1978.

[2] S. H. Fuller, A. K. Jones, I. Durham, eds.
Cm review.*
Technical Report, Computer Science Department, Carnegie-Mellon University, June, 1977.

[3] S. H. Fuller, J. K. Ousterhout, L. Raskin, P. Rubinfeld, P. S. Sindhu, and R. J. Swan.
Multi-microprocessors: An overview and working example.
Proceedings of the IEEE 66(2):216 – 28, February, 1978.

[4] A. K. Jones, R. J. Chansler, Jr., I. Durham, P. Feiler, and K. Schwans.
Software management of Cm*—a distributed multiprocessor.
In *National Computer Conference, Proceedings, Vol. 46*, pages 657 – 63. AFIPS, 1977.

[5] A. K. Jones, R. J. Chansler, Jr., I. Durham, P. Feiler, D. A. Scelza, K. Schwans, and S. R. Vegdahl.
Programming issues raised by a multiprocessor.
Proceedings of the IEEE 66(2):229 – 37, February, 1978.

[6] A. K. Jones, Robert J. Chansler, Jr., I. Durham, K. Schwans, and S. R. Vegdahl.
StarOS, a multiprocessor operating system for the support of task forces.
In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 117 – 27. ACM/SIGOPS, Pacific Grove, California, December 10 – 12, 1979.

[7] A. K. Jones and K. Schwans.
TASK forces: distributed software for solving problems of substantial size.
In *4th International Conference on Software Engineering*, pages 315 – 30. ACM/SIGSOFT, Munich, Germany, September, 1979.

[8] A. K. Jones and P. Schwarz.
Experience using multiprocessor systems: a status report.
Computing Surveys 11(2), June, 1980.
An earlier version is available as CMU-CS-79-146 from the Computer Science Department, Carnegie-Mellon University.

[9] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu.
Medusa: an experiment in distributed operating system structure.
Communications of the ACM 23(2), February, 1980.

[10] John K. Ousterhout.
Partitioning and cooperation in a distributed multiprocessor operating system: Medusa.
PhD thesis, Carnegie-Mellon University, April, 1980.

[11] Levy Raskin.
Performance evaluation of multiple processor systems.

PhD thesis, Carnegie-Mellon University, August, 1978.
Available as CMU tech report CMU-CS-78-141.

[12] D. P. Siewiorek, V. Kini, H. Mashburn, S. R. McConnel, and M. Tsao.
A case study of C.mmp, Cm*, and C.vmp: Part I—experiences with fault tolerance in multiprocessor systems.
Proceedings of the IEEE 66(10):1178 – 99, October, 1978.

[13] D. P. Siewiorek, V. Kini, R. Joobbani, and H. Bellis.
A case study of C.mmp, Cm*, and C.vmp: Part II—predicting and calibrating reliability of multiprocessor systems.
Proceedings of the IEEE 66(10):1200 – 20, October, 1978.

[14] R. J. Swan, S. H. Fuller, and D. P. Siewiorek.
Cm*—a modular, multi-microprocessor.
In *National Computer Conference, Proceedings, Vol. 46*, pages 637 – 44. AFIPS, 1977.

[15] R. J. Swan, A. Bechtolsheim, K. W. Lai, and J. K. Ousterhout.
The implementation of the Cm* multi-microprocessor.
In *National Computer Conference, Proceedings, Vol. 46*, pages 645 – 55. AFIPS, 1977.

[16] R. J. Swan.
*The switching structure and addressing architecture of an extensible multiprocessor, Cm**.
PhD thesis, Carnegie-Mellon University, August, 1978.

[17] M. M. Tsao.
A study of transient errors on Cm*.
Master's thesis, Carnegie-MellonUniversity, December, 1978.

Appendix II
The Cm* Family

These people have contributed to Cm* in one way or another:

Bechtolsheim, Andreas	Fuller, Samuel	Kazar, Michael	Rubinfeld, Paul
Bellis, Harold	Gasbarro, James	Lai, Kwok-Woon	Scelza, Donald
Brantley, William	Gehringer, Edward	Lebovitz, Gregg	Schwans, Karsten
Butcher, Lawrence	Gosling, James	Lesser, Victor	Siewiorek, Daniel
Carey, Michael	Gramlich, Wayne	Mohan, Joseph	Sindhu, Pradeep
Chansler, Robert	Grosser, John	Ostlund, Neil	Sproull, Robert
Coraluppi, Paolo	Gupta, Satish	Ousterhout, John	Swan, Richard
Deminet, Jaroslaw	Hibbard, Peter	Raskin, Levy	Tsao, Michael
Durham, Ivor	Hisgen, Andrew	Reddy, Pradeep	Van Zoeren, Harold
Feiler, Peter	Jones, Anita	Rodeheffer, Thomas	Vegdahl, Steven

The following people contributed to the Computer Modules project, the precursor of Cm*:

Bell, Gordon	Clark, Douglas	Rege, Satish	Thomas, Donald
Chen, Robert	Grason, John		

In addition, we acknowledge the assistance of the following people from DEC:

Dicket, Duane	Downey, Bobbi	Teicher, Stephen	Titelbaum, Michael
Dickman, Lloyd	Olsen, Richard		

Acknowledgments

This research was supported by the Defense Advanced Research Projects Agency and the Digital Equipment Corporation. Ivor Durham received support from an IBM fellowship, Steven Vegdahl from the Hertz Foundation, and John Ousterhout from a National Science Foundation fellowship. Jaroslaw Deminet is on leave from the University of Warsaw.

Index